Vdbench
Users Guide

Version: 5.04.06

July 2016

Author: Henk Vandenbergh

1. Copyright Notice

2. Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

3. License Restrictions Warranty/Consequential Damages Disclaimer

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

4. Warranty Disclaimer

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

5. Restricted Rights Notice

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

6. Hazardous Applications Notice

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate failsafe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

7. Third Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible

for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

TABLE OF CONTENTS :

1 Vdbench: Disk I/O Workload Generator

Getting started with Vdbench:

1.1 Introduction

Vdbench is a disk I/O workload generator to be used for testing and benchmarking of existing and future storage products.

Vdbench is written in Java with the objective of supporting Oracle heterogeneous attachment. At this time I/O has been tested on Solaris Sparc and x86, All flavors of Windows, HP/UX, AIX, Linux, Mac OS X, zLinux and RaspBerry Pi.

Note: one or more of these platforms may not be available for this latest release, this due to the fact that a proper system for a Java JNI C compile may not have been available at the time of distribution. In this case there will be a 'readme.txt' file in the OS specific subdirectory, asking for a volunteer to do a small Java JNI C compile.

1.2 Objective

The objective of Vdbench is to generate a wide variety of controlled storage I/O workloads, allowing control over workload parameters such as I/O rate, LUN or file sizes, transfer sizes, thread count, volume count, volume skew, read/write ratios, read and write cache hit percentages, and random or sequential workloads. This applies to both raw disks and file system files and is integrated with a detailed performance reporting mechanism eliminating the need for the Solaris command *iostat* or equivalent performance reporting tools. I/O performance reports are web accessible and are linked using HTML. Just point your browser to the *summary.html* file in the Vdbench output directory.

There is no requirement for Vdbench to run as root as long as the user has read/write access for the target disk(s) or file system(s) and for the output-reporting directory.

Non-performance related functionality includes Data Validation with Vdbench keeping track of what data is written where, allowing validation after either a controlled or uncontrolled shutdown.

1.3 Terminology

- ***Execution*** parameters control the overall execution of Vdbench and control things like parameter file name and target output directory name.
- ***Raw I/O workload*** parameters describe the storage configuration to be used and the workload to be generated. The parameters include ***General, Host Definition (HD), Replay Group (RG), Storage Definition (SD), Workload Definition (WD)*** and ***Run Definition (RD)*** and must always be entered in the order in which they are listed here**.** A ***Run*** is the execution of one workload requested by a Run Definition. Multiple ***Runs*** can be requested within one Run Definition.
- ***File system Workload*** parameters describe the file system configuration to be used and the workload to be generated. The parameters include ***General, Host Definition (HD), File System Definition (FSD), File system Workload Definition (FWD)*** and ***Run Definition (RD)*** and must always be entered in the order in which they are listed here. A ***Run*** is the execution of one workload requested by a Run Definition. Multiple ***Runs*** can be requested within one Run Definition.
- ***Replay***: This Vdbench function will replay the I/O workload traced with and processed by the Sun StorageTek™ Workload Analysis Tool (Swat).
- ***Master*** and ***Slave***: Vdbench runs as two or more Java Virtual Machines (JVMs). The JVM that you start is the master. The master takes care of the parsing of all the parameters, it determines which workloads should run, and then will also do all the reporting. The actual workload is executed by one or more Slaves. A Slave can run on the host where the Master was started, or it can run on any remote host as defined in the parameter file. See also '-m nn': Multi JVM Execution
- ***Data Validation:*** Though the main objective of Vdbench has always been to execute storage I/O workloads, Vdbench also is very good at identifying data corruptions on your storage.
- ***Journaling:*** A combination of Data Validation and Journaling allows you to identify data corruption issues across executions of Vdbench. See Data Validation and Journaling.
- LBA, or lba: For Vdbench this never means Logical **Block** Address, it is Logical **Byte** Address. 16 years ago I decided that I did not want to have to worry about disk sector size changes, and it is clear that this was the right decision.

1.4 Installing Vdbench

Vdbench is packaged as a *zip* file. Unzip the file and you're ready to go.

The zip file contains everything you need for both Windows and Unix systems.

Note: one or more of the many supported platforms may not be available for this latest release, this due to the fact that a proper system for a Java JNI C compile may not have been available at the time of distribution. In this case there will be a 'readme.txt' file in the OS specific subdirectory, asking for a volunteer to do a small Java JNI C compile.

1.5 How to start Vdbench:

You can do a very quick simple test without even having to create a parameter file:

- ./vdbench –t (for a raw I/O workload)
- ./vdbench -tf (for a file system workload)

After this, use your favorite web browser to look at /vdbench/output/summary.html and you'll see the reports that Vdbench creates.

To start Vdbench:

- Unix: /home/vdbench/vdbench –f parmfile
- Windows: c:\vdbench\vdbench.bat –f parmfile

You can find some simple example parameter files here: **sample parameter files.**
There are many more examples in the ../examples/ directory.

1.6 Execution parameter overview

./vdbench [- fxxx yyy zzz] [-o xxx] [-c x] [-s] [-k] [-e nn] [-I nn] [-w nn] [-m nn] [-v] [-vr] [-vw] [-vt] [-j] [-jr] [-jri] [-jm] [-jn] [-jro] [-p nnn] [-t] [-l nnn] [xxx=yyy,….]

Execution parameters must be specified individually: Enter '-v –fparmfile', and not '-vfparmfile'.

or,
for some Vdbench utility functions:

./vdbench [compare] [csim] [dsim] [dvpost] [edit] [jstack] [parse] [print] [rsh] [sds] [showlba]

(dvpost may not be in this new release, but may come back. The reference to dvpost therefore will stay in this document until the final decision has been made),

Here is a brief description of each parameter, with a link to a more detailed description:

1.6.1 Execution Parameters

See also Execution Parameter Detail.

compare	Start Vdbench workload compare
csim	Compression simulator
dsim	Dedup simulator
dvpost	Post-processing of output generated by Data Validation
edit	Primitive full screen editor, syntax './vdbench edit file.name'.
jstack	Create stack trace. Requires a JDK.
parse(flat)	Selective parsing of flatfile.html
print	Print any block on any disk or disk file
rsh	Start Vdbench RSH daemon (For multi-host testing)
sds	Start Vdbench SD parameter generation tool (Solaris, Linux, Windows)
showlba	Used to display output of the XXXX parameter.
-f xxx yyy zzz	Workload parameter file name(s). One parameter file is required.
-o xxx	Output directory for reporting. Default is 'output' in current directory.
-t	Run a five second sample workload on a small disk file (for demo).
-tf	Run a five second sample File system workload.

-e nn	Override 'elapsed' parameters in Run Definitions (RD)
-I nn	Override 'interval' parameters in Run Definitions (RD)
-w nn	Override 'warmup' parameters in Run Definitions (RD).
-m nn	Override the amount of concurrent JVMs to run workload
-v	Activate data validation.
-vr	Activate data validation, immediately re-read after each write.
-vw	Activate data validation, but don't read before write.
-vt	Activate data validation, keep track of each write timestamp (memory intensive)
-j	Activate data validation with journaling.
-jr	Recover existing journal, validate data and run workload
-jro	Recover existing journal, validate data but do not run requested workload.
-jri	Recover existing journal, ignoring pending writes.
-jm	Activate journaling, but only write the journal maps.
-jn	Activate journaling, but use asynchronous writes to journal.
-s	Simulate execution. Scans parameter files and displays run names.
-k	Solaris only: Report kstat statistics on console.
-c	Clean (delete) existing FSD file system structure at start of run.
-co	Force format=only
-cy	Force format=yes
-cn	Force format=no
-p nnn	Override Java socket port number (default 5570).
-l nnn	(lowercase 'L'): After the last run, start over with the first run. Without *nnn* this is an endless loop, or loop for a total nnn s/m/h seconds/minutes/hours, e.g. -l 24h. See also 'loop=nn' general parameter
xxx=yyy …..	See variable substitution.

1.7 Parameter File(s)

The parameter files entered will be read in the order specified. All parameters have a required order as defined here: **General**, **HD, RG, SD, WD** and **RD**, or for file system testing: **General**, **HD, FSD, FWD** and **RD.**
Note that not all types of parameters are always needed.

1.7.1 Variable substitution.

Variable substitution allows you to code variables like $lun in your parameter file which then can be overridden from the command line. For example:
sd=sd1,lun=$lun
$lun must be overridden from the command line: ./vdbench -f parmfile lun=/dev/x.

In case your parameter file is embedded in a shell script, you may also specify a '!' to prevent accidental substitution by the scripting language, e.g. sd=sd1,lun=!lun

Note that any variable specified on the command line but not found in the parameter file will caused Vdbench to abort.

1.7.2 Multi-host parameter replication.

Whenever the constant '$host', '!host' or '#host' is found in an input line in a parameter file, this line is automatically repeated once for each host label that has been defined using the Host Definition (HD) parameters. Some times when you run tests against multiple different hosts, directing file system workloads towards specific target hosts can become mighty complex. The $host parameter is there to make life a little easier. A simple example:

hd=host1,… hd=host2,…. fsd=fsd_$host,anchor=/dir/$host,…..	Result: fsd=fsd_host1,anchor=/dir/host1,….. fsd=fsd_host2,anchor=/dir/host2,…..
Just add host=host3,….	fsd=fsd_host1,anchor=/dir/host1,….. fsd=fsd_host2,anchor=/dir/host2,….. fsd=fsd_host3,anchor=/dir/host3,…..

Note that this only works on one single line in the parameter file, not if the parameters are split over multiple lines, for instance using above example, one line for fsd=fsd_$host, and then anchor= on the next line.

'$host' and '!host' are replaced with the host label. '!host' is there to prevent problems when you include your parameter file inside of a shell script that is trying to interpret $host too early.
'#host' is replaced with the current relative host, 0,1,2, etc.

1.7.3 include=parmfile

There is however one parameter that can be anywhere: include=/parm/file/name
When this parameter is found, the contents of the file name specified will be copied in place.
Example: include=/complicated/workload/definitions.txt

You can use as many includes as needed, though overuse of this parameter will make it very difficult to take a quick look at a parameter file to see what's being requested. File 'parmfile.html' in the output directory will show you the final results of everything that has been included.

1.7.4 General Parameters: Overview

These parameters must be the ***first*** parameters in the parameter file, before any HD, SD or FSD.
See also General Parameter Detail.

General parameters	
abort_failed_skew=nnn	Abort if requested workload skew is off by more than nnn%
compratio=nn	Specify the compression ratio of the data pattern used for writes.
concatenatesds=yes	See SD Concatenation
create_anchors=yes	Create parent directories for FSD anchor.
data_errors=nn	Terminate after 'nn' read/write/data validation errors (default 50)
data_errors=cmd	Run command or script 'cmd' after first read/write/data validation error, then terminate.
See also Data Deduplication parameters:	
dedupratio=	Expected ratio. Default 1 (all blocks are unique).
dedupunit=	What size of data does Dedup compare?
dedupsets=	How many sets of duplicates.
deduphotsets=	Dedicated small sets of duplicates
dedupflipflop=	Activate the Dedup flip-flop logic.
endcmd=cmd	Execute command or script at the end of the last run
formatsds=	Force a one-time (pre)format of all SDs
formatxfersize=	Specify xfersize used when creating, expanding, or (pre)formatting an SD.
histogram=(default,….)	Override defaults for response time histogram.
include=/file/name	Includes /file/name inline. See above.
loop=	Repeat all Run Definitions: loop=nn repeat nn times

	loop=nn[s|m|h] repeat until nn seconds/minutes/hours See also '-l nn' execution parameter
Journaling parameters:	
journal=yes	Activate Data Validation and Journaling:
journal=recover	Recover existing journal, validate data and run workload
journal=only	Recover existing journal, validate data but do not run requested workload.
journal=noflush	Use asynchronous I/O on journal files
journal=maponly	Do NOT write before/after journal records
journal=skip_read_all	After journal recovery, do NO read and validate every data block.
journal=(max=nnn)	Prevent the journal file from getting larger than nnn bytes
journal=ignore_pending	Ignore pending writes during journal recovery.
messagescan=no	Do not scan /var/xxx/messages (Solaris or Linux)
messagescan=nodisplay	Scan but do not display on console, instead display on slave's stdout.
messagescan=nnnn	Scan, but do not report more than nnn lines. Default 1000
monitor=/file/name	See External control of Vdbench termination
pattern=	Override the default data pattern generation.
port=nn	Override the Java socket port number.
report=host_detail report=slave_detail	Specifies which SD detail reports to generate. Default is SD total only.
report=no_sd_detail	Will suppress the creation of SD specific reports.
report_run_totals=yes	Reports run totals.
startcmd=cmd	Execute command or script at the beginning of the first run
showlba=yes	Create a 'trace' file so serve as input to ./vdbench showlba
Data Validation parameters:	
validate=yes	(-vt) Activate Data Validation. Options can be combined: validate=(x,y,z)
validate=read_after_write	(-vr) Re-reads a data block immediately after it was written.
validate=no_preread	(-vw) Do not read before rewrite, though this defeats the purpose of data validation!
validate=time	(-vt) keep track of each write timestamp (memory intensive)

1.7.5 Host Definition (HD) Parameter overview

These parameters are ONLY needed when running Vdbench in a multi-host environment or if you want to override the number of JVMs used in a single-host environment.
See also Host Definition parameter detail.

hd=default	Sets defaults for all HDs that are entered later
hd=localhost	Sets values for the current host
hd=host_label	Specify a host label.
system=host_name	Host IP address or network name, e.g. xyz.customer.com
vdbench=vdbench_dir_name	Where to find Vdbench on a remote host if different from current.
jvms=nnn	How many slaves to use. See Multi JVM execution.
shell=rsh | ssh | vdbench	How to start a Vdbench slave on a remote system.
user=xxxx	Userid on remote system Required.
clients=nn	This host will simulate the running of multiple 'clients'. Very useful if you want to simulate numerous clients for file servers without having all the hardware.
mount="mount xxx …"	This mount command is issued on the target host after the possibly needed mount directories have been created.

1.7.6 Replay Group (RG) Parameter Overview

See also Swat and Vdbench Replay

rg=name	Unique name for this Replay Group (RG).
devices=(xxx,yyy,….)	The device numbers from Swat's flatfile.bin.gz to be replayed.

Example: rg=group1,devices=(89465200,6568108,110)

Note: Swat Trace Facility (STF) will create Replay parameters for you. Select the 'File' 'Create Replay parameter file' menu option. All that's then left to do is specify enough SDs to satisfy the amount of gigabytes needed.

1.7.7 Storage Definition (SD) Parameter Overview

See also Storage Definition Parameter Detail.
This set of parameters identifies each physical or logical volume manager volume or file system file used in the requested workload. Of course, with a file system file, the file system takes the responsibility of all I/O: reads and writes can and will be cached (see also openflags=) and Vdbench will not have control over *physical* I/O. However, Vdbench can be used to test file system file performance (See also File system testing).

Example: sd=sd1,lun=/dev/rdsk/cxt0d0s0,threads=8

sd=default	Sets defaults for all SDs that are entered later.
sd=name	Unique name for this Storage Definition (SD).
count=(nn,mm)	Creates a sequence of SD parameters.
align=nnn	Generate logical byte address in 'nnn' byte boundaries, not using default 'xfersize' boundaries.
dedupratio=	See data deduplication:
dedupsets=	
deduphotsets=	
dedupflipflop=	
hitarea=nn	See read hit percentage for an explanation. Default 1m.
host=name	Name of host where this SD can be found. Default 'localhost'
journal=xxx	Directory name for journal file for data validation
lun=lun_name	Name of raw disk or file system file.
offset=nnn	At which offset in a lun to start I/O.
openflags=(flag,..)	Pass specific flags when opening a lun or file
range=(nn,mm)	Use only a subset 'range=nn': Limit Seek Range of this SD.
replay=(group,..)	Replay Group(s) using this SD.
replay=(nnn,..)	Device number(s) to select for Swat Vdbench replay
resetbus=nnn	Issue ioctl (USCSI_RESET_ALL) every nnn seconds. Solaris only
resetlun=nnn	Issue ioctl (USCSI_RESET) every nnn seconds. Solaris only
size=nn	Size of the raw disk or file to use for workload. Optional unless you want Vdbench to create a disk file for you.
threads=nn	Maximum number of concurrent outstanding I/O for this SD. Default 8

1.7.8 File system Definition (FSD) Parameter Overview

See Filesystem Definition (FSD) parameter overview

1.7.9 Workload Definition (WD) Parameter Overview

See also Workload Definition Parameter Detail.
The Workload Definition parameters describe what kind of workload must be executed using the storage definitions entered.

Example: wd=wd1,sd=(sd1,sd2),rdpct=100,xfersize=4k

wd=default	Sets defaults for all WDs that are entered later.
wd=name	Unique name for this Workload Definition (WD)
sd=xx	Name(s) of Storage Definition(s) to use
host=host_label	Which host to run this workload on. Default localhost.
hotband=	See hotbanding
iorate=nn	Requested fixed I/O rate for this workload.
openflags=(flag,..)	Pass specific flags when opening a lun or file.
priority=nn	I/O priority to be used for this workload.
range=(nn,nn)	Limit seek range to a defined range within an SD.
rdpct=nn	Read percentage. Default 100.
rhpct=nn	Read hit percentage. Default 0.
seekpct=nn	Percentage of random seeks. Default seekpct=100 or seekpct=random.
skew=nn	Percentage of skew that this workload receives from the total I/O rate.
streams=(nn,mm)	Create independent sequential streams on the same device.
stride=(min,max)	To allow for skip-sequential I/O.
threads=nn	Only available during SD concatenation.
whpct=nn	Write hit percentage. Default 0.
xfersize=nn	Data transfer size. Default 4k.
xfersize=(n,m,n,m,..)	Specify a distribution list with percentages.
xfersize=(min,max,align)	Generate xfersize as a random value between min and max.

1.7.10 File system Workload Definition (FWD) Parameter Overview

See Filesystem Workload Definition (FWD) parameter overview

1.7.11 Run Definition (RD) Parameter Overview (For raw I/O testing)

See also Run Definition Parameter Detail. For File system testing see: RD for File systems.

The Run Definition parameters define which of the earlier defined workloads need to be executed, what I/O rates need to be generated, and how long the workload will run. One Run Definition can result in multiple actual workloads, depending on the parameters used.

Example: rd=run1,wd=(wd1,wd2),iorate=1000,elapsed=60,interval=5

There is a separate list of RD parameters for file system testing.

rd=default	Sets defaults for all RDs that are entered later.
rd=name	Unique name for this Run Definition (RD).
wd=xx	Workload Definitions to use for this run.
sd=xxx	Which SDs to use for this run (Optional).
curve=(nn,nn,..)	Data points to generate when creating a performance curve. See also stopcurve=
distribution=(x[,variable]	I/O inter arrival time calculations: exponential, uniform, or deterministic. Default exponential.
elapsed=nn	Elapsed time for this run in seconds. Default 30 seconds.
endcmd=cmd	Execute command or script at the end of the last run
(for)compratio=nn	Multiple runs for each compression ratio.
(for)hitarea=nn	Multiple runs for each hit area size.
(for)hpct=nn	Multiple runs for each read hit percentage.
(for)rdpct=nn	Multiple runs for each read percentage.
(for)seekpct=nn	Multiple runs for each seek percentage.
(for)threads=nn	Multiple runs for each thread count.
(for)whpct=nn	Multiple runs for each write hit percentage.
(for)xfersize=nn	Multiple runs for each data transfer size.
Most forxxx parameters may be abbreviated to their regular name, e.g. xfersize=(..,..)	
interval=nn	Stop the run after nnn bytes have been read or written, e.g. maxdata=200g. I/O will stop at the lower of elapsed= and maxdata=.
iorate=(nn,nn,nn,…)	Reporting interval in seconds. Default 'min(elapsed/2,60)'
iorate=curve	One or more I/O rates.
iorate=max	Create a performance curve.
iorate=(nn,ss,…)	Run an ***uncontrolled*** workload.
	nn,ss: pairs of I/O rates and seconds of duration for this I/O rate. See also 'distribution=variable'.

openflags=xxxx	Pass specific flags when opening a lun or file
pause=nn	Sleep 'nn' seconds before starting next run.
replay=(filename, split=split_dir, repeat=nn)	-'filename': Replay file name used for Swat Vdbench replay - 'split_dir': directory used to do the replay file split. - 'nn': how often to repeat the replay.
startcmd=cmd	Execute command or script at the beginning of the first run
stopcurve=n.n	Stop iorate=curve runs when response time > n.n ms.
warmup=nn	Override warmup period.

1.8 Execution parameter detail

1.8.1 '-f xxx ': Workload Parameter File(s)

The workload parameter file(s) contains all the workload parameters.

There are five groups of parameters in the file: General (optional), Host Definition (**HD**) (optional), Storage Definition (**SD**), Workload Definition (**WD**), and Run Definition (**RD**). For File system testing this will be General (optional), Host Definition (**HD**) (optional), File System Definition (**FSD**), File system Workload Definition (**FWD**), and Run Definition (**RD**).
These groups must be entered in the order defined here.

Each parameter has a keyword followed by one or more sub parameters. Most keywords (and alphanumeric sub parameters) can be abbreviated to its shortest unique value with a minimum of two characters. For example xfersize=512 can be abbreviated to xf=512. Sub parameters can be coded with a single value 'iorate=1000', or with multiple values 'iorate=(100,200,300)'. Multiple values must always be enclosed within parentheses. A set of sub parameters must be either numeric, or alphanumeric, not a mix. Not all keywords accept multiple sub parameters, but the documentation will make clear which parameters will accept them. Keywords may be entered in mixed case: e.g. 'Xfersize=4k'. When using embedded blanks or other special characters (',' '-' or '=') you must encapsulate the parameters in double quotes.

Numeric parameters allow definition in (k)ilobytes, (m)egabytes, (g)igabytes and (t)erabytes. k/m/g/t may be specified in lower or upper case. One kilobyte equals 1024 bytes. Time values may also be entered as minutes or hours; e.g., 'elapsed=7200 is equivalent to 'elapsed=120m' or ‘elapsed=2h’.

Multiple numeric values can be entered as follows:

keyword=(1,2,3,4,5,6,7,8,9,10,..)	Individual values
keyword=(1-10,1)	Range, from 1 to 10, incremented by 1 (1,2,3,4,5,6,7,8,9,10)
keyword=(1-64,d)	Doubles: from 1 to 64, each successive value doubled (1,2,4,8,16,32,64)
keyword=(64,1,d)	Reverse double (divide by two)

A detailed parameter scan report is written to *output/parmscan.html*. When there are problems with the interpretation of the keywords and sub parameters, looking at this file can be very helpful because it shows the last parameter that was read and interpreted. A complete copy of the input parameters is written to *output/parmfile.html*. This is done so that when you look at a I/O output directory you will see exactly what workload was executed -- no more guessing trying to remember ‘what did I run 6 months ago’.

Comments: a line starting with '/', '#' or '*' and anything following the first blank on a line (except when inside of double quotes) is considered a comment. Also, a line beginning with 'eof' is treated as end of file, so whatever is beyond that will be ignored.
Continuation: If a line gets too large you may continue the parameters on the next line by ending the line with a comma and a blank and then start the next line with a new keyword.

Vdbench allows for the specification of multiple parameter files. This allows for instance the separation of SD parameters from WD and RD parameters. Imagine running the same workload on different storage configurations. You can then create one WD and RD parameter file, and multiple SD parameter files, running it as follows: "./vdbench –f sd_parmfile wd_rd_parmfile".

include=/parm/file/name

When this parameter is found, the contents of the file name specified will be copied in place.
Example: include=/complicated/workload/definitions.parmfile

See also variable substitution.

1.8.2 '-oxxx': Output Directory

Vdbench writes all its HTML files to this output directory. The directory will be created if it does not exist. An already existing directory will be reused after first deleting all existing HTML files. Reused directories can contain HTML files that were placed there by an earlier execution of Vdbench, and may not be related in any way to the new execution. Leaving these files around could cause confusion, so the old HTML files are deleted.
If you do not want to reuse an output directory, you may add a '+' after the directory name: e.g. '-o dirname+' . If 'dirname' does not exist it will be created. If it *does* exist, the directory name is incremented by one to 'dirname001', and if that directory name is available it is created. And so on until 'dirname999', after which Vdbench will stop.
You may also request that a timestamp be added to the output directory name: '-o output.tod' will result in a directory named 'output.yymmdd.hhmmss'.
This dynamic creation of new output directory names is very useful if you don't want to accidentally overwrite this very important test that just took you 24 hours. ☺

Default: '-f output'.

1.8.3 '-v': Activate Data Validation

This execution parameter activates Data Validation. Each write of a block will be recorded, and after the next read to the same block, the block's old contents will be validated. The next write to the block will cause the block to be read first and then validated.
Option '-vr' can be used to do a read and validate immediately after each write, versus normally only validating when a block of data is scheduled for the next read or the next write. When doing I/O against a large LUN it can normally take quite a while before a block is referenced again. So, at times '-vr' may be useful to get a quick confirmation that the data is correct.

Be aware, however, that reading a block immediately after a write likely will only show that the data reached the file system or storage controller cache and there is no proof that the data ever reached the physical disk drives.
Option '-vt' will save the timestamp of the last successful read or write in memory (no journaling available). When that data block fails this timestamp will be reported. Knowing the time of day that the block was good can help you identify which error injection might have caused the problem. Beware: this requires 8 more bytes of java heap memory per data block (memory needs for 512 byte blocks could therefore be prohibitive). An internal limit for this is set to allow for now more than 31bits of blocks, or about 16 gigabytes of java heap memory.
See Data Validation and Journaling for a more detailed description of data validation.

Data Validation can also be activated using the 'validate=yes' parameter in the parameter file.

1.8.4 '-j': Activate Data Validation and Journaling

Journaling allows data validation to continue after Vdbench or the operating system terminates. '-j' creates a new journal file or overwrites an existing one. Specify '-jr' to recover an existing journal. '-jn' prevents a flush to disk on journal writes (by default, each journal write is flushed directly to disk(synchronous I/O)). However, be aware that if the operating system terminates without a proper shutdown, the unflushed journal file may be incomplete.
See Data Validation and Journaling for a more detailed description of data validation.

Journaling can also be activated using the 'journal=yes' parameter in the parameter file.

1.8.5 '-s': Simulate Execution

Vdbench can create large and complex workloads. A simulation run scans and interprets the parameter file(s), but does not execute the workloads.

1.8.6 '-k': Kstat Statistics on Console

On Solaris systems, kstat performance statistics are reported to *kstat.html*. To allow these statistics also to be written to the active console window, specify '-k'.
Vdbench does its utmost to match the requested LUN and/or file names with correct Kstat information. Veritas VxVm, QFS, SVM, and ZFS are supported, but there are situations where Vdbench has some problems. When Vdbench fails to find the correct Kstat information, execution continues, but without using Kstat.

Solaris device names (/dev/rdsk/cxtxdxsx) are translated to Kstat instance names using the output of iostat. Output of 'iostat-xd' is matched with output of 'iostat –xdn', and the device and instance names are taken from there. If a device name cannot be translated to the proper Kstat instance name this way there is possibly a bug in Solaris that needs to be resolved.

1.8.7 '-m nn': Multi JVM Execution

Depending on the processor speed, there is a maximum number of IOPS or maximum thread count that a single Java Virtual Machine (JVM) can handle. Since Java runs as a single process, it is bound by what a single process can do. To alleviate this problem, Vdbench starts an extra copy of itself (a slave) for each requested 100,000.
The maximum number of JVMs started this way is limited by the number specified with the '-m nn' execution parameter or 'hd=default,jvms=nn', or the lower of (# SDs or 8).

This maximum default of 8 JVMs can become a problem if the system that you are testing on is small and low on memory. If you know that your maximum IOPS will stay below 100,000 per JVM you should set your JVM count to whatever is needed, not more.

Each workload is executed on each JVM or slave, except for sequential workloads. Running sequential workloads on each slave would result in the same sequential blocks being read by each slave, something that makes for nice performance numbers, but that does not really represent an accurate sequential workload. Sequential workloads therefore are spread round robin over each available JVM/slave.

The JVM count can also be set (and that is the preferred method) using the hd=default,jvms=nn or hd=hostX,jvms=nn parameter.

Vdbench will display a warning when more than 100,000 iops are done per JVM. This serves as a warning that you possibly do not have enough active JVMs and you may have to experiment increasing this value. Note that if a lot of these IOPS result from file system cache hits this default limit of 100,000 per JVM could be much higher.

1.8.8 '-t' or '-tf': Sample Vdbench execution.

When running './vdbench –t' Vdbench will run a hard-coded sample run. A small temporary file is created and a 50/50 read/write test is executed for just five seconds.
This is a great way to test that Vdbench has been correctly installed and works for the current OS platform without the need to first create a parameter file.

./vdbench -tf will run a quick File system test.

1.8.9 '-e nn' Override elapsed time

This parameter can be used to temporarily override the value of any elapsed= parameters specified in the parameter file. This can be very useful to quickly discover problems. For instance, you just created a 24hour test run, including multiple Run Definitions (RDs). If there is a problem you don't want to find out about after hours and hours of running. Just run the test with only a few seconds or minutes of elapsed time to see how things work for you.

1.8.10 '-i nn' Override report interval time.

This overrides all interval= values specified in the parameter file. See also '-e nn' above.

1.8.11‘-w nn’ Override warmup time.

This overrides all warmup= values specified in the parameter file. See also ‘-e nn’ above.

1.9 Vdbench utility functions.

Vdbench has several small utility functions to help you with your day-to-day Vdbench testing. Just enter ‘./vdbench xxx’, for instance ./vdbench sds’.

1.9.1 ./vdbench sds: Generate Vdbench SD parameters.

Sick and tired of entering 200 50+ hexadecimal Solaris device names without any typos? Run ./vdbench sds and Vdbench will do it for you.
See Vdbench SD parameter generation tool. This works for Solaris, Linux and Windows,

1.9.2 ./vdbench dvpost: Data Validation post processing

DvPost as of vdbench50405 no longer exists. However, that decision may still be reversed, depending on the feedback that I receive. For now I'll keep it in the documentation.

Running ‘./vdbench dvpost’ brings up a screen that allows you to zoom in on the Data Validation errorlog.html file. This allows you to ‘quickly’ skim through possibly thousands and thousands of lines of data generated when a data corruption is recognized by Vdbench. See also Vdbench Data Validation post-processing tool.

1.9.3 ./vdbench jstack: Display java execution stacks of active Vdbench runs.

Like all software, there always is a chance that there is a problem or bug. Code hangs are very difficult to fix if you don’t know where the problem is. ‘./vdbench jstack’ will print out the active Java execution stack of all currently running Java programs. Run this before killing Vdbench.
Must be run using the same user id used for Vdbench; you also must be using a JDK/SDK, since that includes the java jstack executable.

1.9.4 ./vdbench rsh: Vdbench RSH daemon.

Not every OS has an RSH or SSH daemon available (windows for instance) for testing, and some times getting the available RSH or SSH to do what you want just does not work.
For those situations, Vdbench has his own (primitive) RSH daemon. It only works for Vdbench. Just run ./vdbench rsh’ once on the target system, and Vdbench will open a java socket that will be used to start Vdbench slaves on that host and return its stdout and stderr output.

1.9.5 ./vdbench print: Print any block on any lun or file.

Especially when running into data corruption issues identified by Data Validation, being able to see what the current contents is of a data block can be very useful.
Syntax: ./vdbench print device lba xfersize

device:	any device or file name.
lba:	logical byte address. May be prefixed with 0x if this is hexadecimal. You may also specify k or m for kilobytes or megabytes.
xfersize:	length of block to print.

1.9.6 ./vdbench edit: Simple full screen editor, or 'back to the future'.

It was around 1978 that I went away from line editors, never expecting to have to go back there. When starting to work with Unix systems back in 2000 all I found was vi. By that time I had lost all my Neanderthal habits, so vi just wasn't the way to go for me.
It took me less than half an hour to write a full screen editor using Java, and here it is: Just run ./vdbench edit /file/name and the convenience of the 21st century will be with you.☺

1.9.7 ./vdbench compare: Compare Vdbench test results.

This function compares two sets of Vdbench output directories and shows the delta iops and response time and optionally the data rate in 9 different colors: light green is good, dark green is better, red is bad, etc. See also: Vdbench Workload Compare.
There is a much more complete Workload Compare available in Swat.

1.9.8 ./vdbench parse: Parse Vdbench flatfile.

The Vdbench flatfile parser is a simple program that takes the flatfile, picks out the columns and rows that the user wants, and then writes it to a tab delimited file. See Vdbench flatfile selective parsing.

1.9.9 ./vdbench csim: Compression simulator

This utility will read an x% sample of a lun or a bunch of files, calls the java implementation of gzip, and will report average compression ratio.

./vdbench csim
Usage: ./csim [-l nnn] [-p nnn] [-x nnn] [-s nnn] disk1, disk2, file1, file2, ...

-l nnn: gzip compression level to use, default 1
-p nnn: which percentage of data to read, default 0.1%
-s nnn: subset percentage. e.g. -s10 reports compression for each 10% of the volume
-u nnn: transfer size unit in bytes for blocks to be read and compressed. Default 128k
disk1, file1, ...: up to 10000 disk or file names or windows drive letters (c)

And yes, maybe some day I'll do this sampling using multi-threading… ☺

1.9.10 ./vdbench dsim: Dedup simulator

This utility reads a lun or some files, creates a hash of the data blocks, stores the resulting hash into a table, and counts the amount of time that a duplicate block has been identified.
This tool was written to help me verify that the Dedup data pattern generated by Vdbench were correct, and likely won't be able to handle too huge an amount of data.
Each unique data block needs about 100 bytes of java heap memory, so if you have 400gb worth of 4k unique blocks to analyze you'll need about 10gb of java heap space.

./vdbench dsim
Usage: ./dsim [-n sss] [-w nn] [-f nn] [-u nnnk] disk1, disk2, dir1, dir2, file1, file2, ...
Where:
-u nnn: dedup unit, amount of bytes to be used for deduplication
-n sss: Notify about progress every 'sss' seconds, default 60.
-w nnn: How many 'worker threads' hashing a block. Default 4.
-f nnn: How many 'lun/file threads' reading luns and files. Default 2.
disk1, file1, ...: up to 10000 disk or file names or windows drive letters (c)

Each 'lun/file thread' reads a one mb block and has 'worker threads' threads hashing each 'dedup unit' piece of data.
While csim can save resources by just sampling a lun, for dedup that won't work, so dedup needs to process the whole lun.

1.9.11./vdbench printjournal: print (subset of) .jnl file

This utility may be of help while trying to understand the history of a corrupted data block, a corruption discovered when journaling is active. The whole journal will be printed if no lba has been specified.

Usage:

```
./vdbench printjournal xxx.jnl [-l nnn]
          xxx.jnl:      your .jnl file
          -l 4096       logical byte address
          -l 4k         k/m/g logical byte address
          -l 0x1000     Hex logical byte address
```


1.9.12 ./vdbench showlba: Visualize data access pattern

This utility requires the use of the 'showlba=yes' parameter. This parameter causes each Vdbench slave to create a trace file that then can be read and displayed by the showlba utility. A sample screenshot below, the result of the following small test:

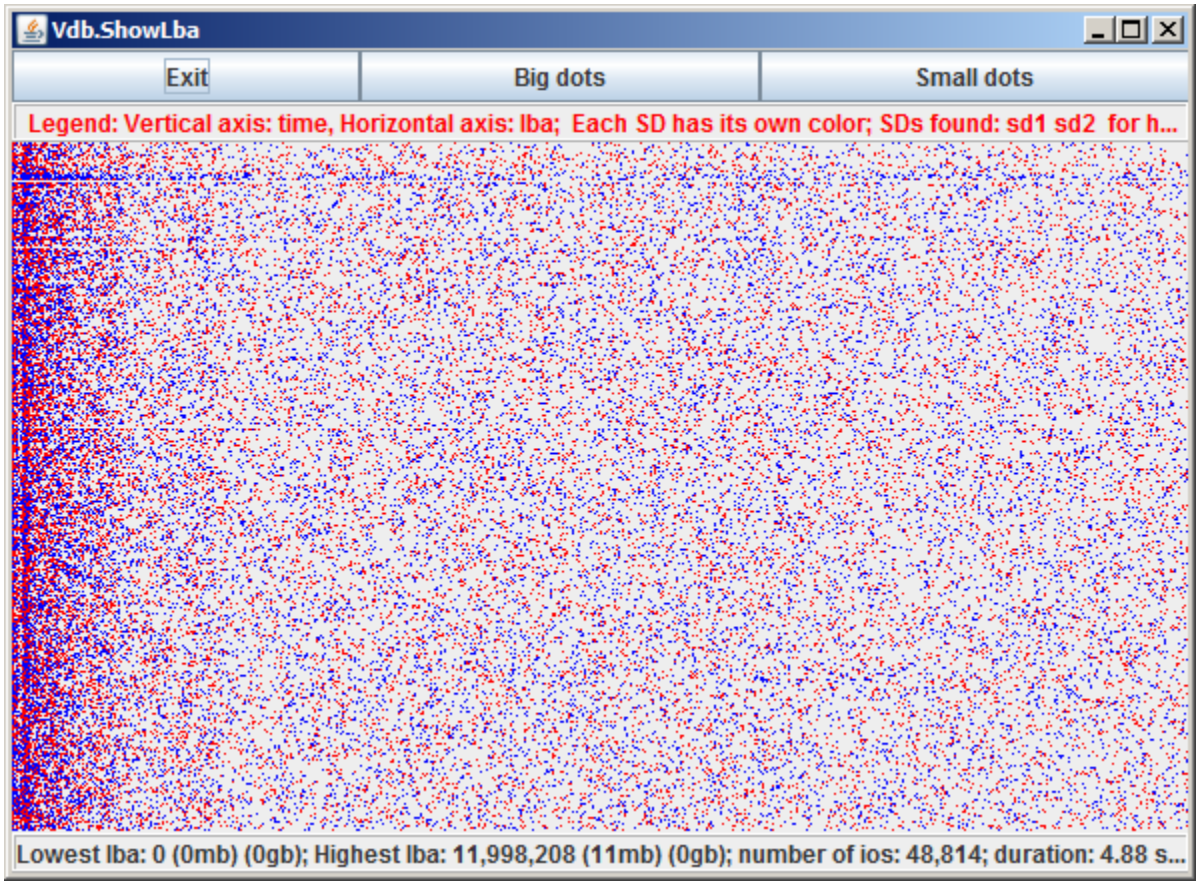
showlba=yes
sd=sd1,lun=w:\temp\quick_vdbench_test,size=40m
sd=sd2,lun=w:\temp\quick_vdbench_test,size=40m
wd=wd1,sd=sd,xf=1k,rdpct=50,hotband=(0,30)*
rd=rd1,wd=wd1,iorate=20000,elapsed=5,interval=1

Note: this is primitive, but it works. ☺ You can find '*showlba.txt' files in your current directory.

Note: please realize that writing this trace file may slow down your workload, and running 24 hours at one million iops may also cause you some problems, so be careful with this.

Usage: ./vdbench showlba [-w WDname] [-s SDname] [-o output.png] *.showlba.txt
Where:

```
-s SDname:       Filter looking for sd=SDname
-w WDname:       Filter looking for wd=WDname
-o output.png    Do not display, create PNG file instead.
*.showlba.txt  . . . .  trace file(s) created by showlba=yes
```



1.10 General parameter detail

1.10.1 'include=parmfile'

As specified earlier, there is a specific order in which these parameters must be specified, with the 'General Parameters' at the beginning of the parameter file.

There is however ONE exception: the 'include=*parmfile'* parameter. This can reside anywhere. It behaves like the good old fashioned #include statement for C code.
The file name specified will be inserted in-line into what is currently being read. You can use as many include= parameters needed.

One use of this parameter could be if you want to keep the SD or FSD parameters separated from the rest of the parameters. SDs and FSDs typically change frequently, while the rest of the parameters stay unchanged.
A different use could be for instance if you have created a fixed, complex 'Application X' set of workloads and you just want to include this existing workload in a new test run.
For instance include=payroll or include=email.

If you code only a file name and not a directory name, Vdbench will look for the file name in the directory of the file containing the current 'include=' statement.

1.10.2 'data_errors=xxx': Terminate After Data Validation or I/O errors

Vdbench by default will abort after 50 data validation or read/write errors. If one or more but less than the specified amount of errors occur, Vdbench at the end of the run (elapsed=) will abort.

There are three ways to define the error count:

data_errors=nn	Causes an abort after nn validation errors. Default 50
data_errors="script_name"	Causes this command/script to be executed after the first error, followed by an abort. This command can be used for diagnostic data collection and/or system dumps.
data_errors=(nn,mm)	Terminates the Vdbench run after 'nn' errors, or 'mm'seconds after the last error. This gives Vdbench the chance to report more errors, but eliminates the possibility that Vdbench keeps running unnoticed for a long period of time after an error.
data_errors=remove	For raw (SD/WD) I/O: The failing SD will be removed but I/O will continue to all others.

The command syntax under ‘data_errors=script_name’ can be as follows:
data_errors= "script_name $output $lun $lba $size". After the error, the script is called with substituted values for 'output directory name', 'LUN name', 'lba', and 'data transfer size’. This allows for quicker collection of diagnostic data.

1.10.3 'startcmd=' and 'endcmd='

Until Vdbench 503 these parameters were known as 'start_cmd' and 'end_cmd' and can continue to be used. Double quotes must be used if any of the commands include a blank.
You may specify one or more commands: startcmd=(cmd1,cmd2,….), each of course using parentheses if needed.

You may also add 'cons', 'sum', or 'log' as an extra parameter, with 'log' being the default. This parameter determines where the output of these commands will be sent: the console, summary.html, or to logfile.html. Example: startcmd=("echo hello world", cons)

(Similar syntax for both startcmd and endcmd)	
startcmd=command1	Run 'command1' on he first slave on a host, sedning output to file '*stdout.html'
startcmd=("xxx yyy",cons)	Run 'xxx yyy' sending output to the console.
startcmd=(cmd,master)	Run 'cmd' on the master
startcmd=(cmd,sum)	Send output to summary.html
startcmd=(cmd,log)	Send output to logfile.html (default)

There are two places where you may use this parameter: Either as a general parameter specified at the beginning of a parameter file, or as part of a Run Definition (RD).

When used as a general parameter, 'startcmd=*cmd'* will be executed right before the first run; 'endcmd=*cmd'* will be executed immediately after the last run is finished or has failed. Whether 'endcmd' will run after a failure of course depends on the type of failure. The use of CTRL-C precludes 'endcmd' from being run.

When used as a Run Definition parameter, 'startcmd=*cmd'* will be executed right before each run; 'endcmd=*cmd'* will be executed immediately after each run is finished.
There is one extra parameter available: 'startcmd=(master,xxx)". By default the command will be run on the first slave on each client, but when you're running on a dozen clients that became a little too expensive. The 'master' parameter will now move the execution to the master JVM, of which there of course is just one.

Note that if you start a long running command you need to add '&' at the end of the command to allow asynchronous execution.

These commands or scripts can be used for anything you like. For example, 'uname –a' and 'psrinfo' come to mind.
A string containing '$output' in the requested command will be replaced by the output directory name requested using the '-o' execution parameter, allowing the command/script access to this directory name.

When starting Vdbench, the command 'config.sh *output.directory*' will always be executed. This allows any kind of preprocessing to be done, or in the default case, the gathering of configuration specific data. After this, 'my_config.sh *output.directory*' will be executed. 'my_config.sh' is there for you to use, and will not be replaced by a (re)installation of Vdbench.
If you never want either of these scripts to be executed, create file 'noconfig' in the Vdbench directory.
Note: Sometimes, depending on which system you are running on, config.sh can take a few seconds or some times minutes. If you don't need this to run, simply create file 'noconfig' in the Vdbench directory.

1.10.4 'pattern=: Data Pattern to be used

By default Vdbench writes a random non-compressible, non dedupable data pattern. See also the 'compratio=' and 'dedupratio=' parameters.

pattern=/file/name	The data found in this file is copied as often as possible into the data pattern buffer.

1.10.5 'compratio=nn': Set compression for data patterns

See also pattern= above.
By default Vdbench will write an uncompressible random data pattern. 'compratio=nn' generates a data pattern that results in a nn:1 ratio.
Understanding that there are different compression algorithms this parameter cannot guarantee the ultimate results. The data patterns implemented are based on the use of the 'LZJB' compression algorithm using a ZFS record size of 128k, and with that in mind the accuracy for this implementation is plus or minus 5%.
This parameter may be overridden using the 'forcompratio=' parameter.

This is how the data pattern is generated:
1. A one-megabyte minimum data pattern buffer is filled with random numbers. This is done only once.
2. A number of random 32-bit words will be replaced with zeroes, this depending on the compratio= parameter used. The number of zeros is based on experiments done using the

LZJB compression algorithm with 128k ZFS recordsize; compression ratios 1:1 through 25:1 are the only ones implemented; any ratio larger than 25:1 will be set to 25.

3. Data starting at the pattern buffer + (remainder of the Logical Byte Address (lba) divided by the length of the pattern buffer) is copied into the data buffer. The copy wraps around to the beginning of the pattern buffer if needed. This takes care of the requested compression ratio.
4. The first 8 bytes of each 4k portion of the data buffer is overlaid by a combination of lba, 4k offset, file handle, and the lower four bytes of the system's High Resolution Timer (HRT). This is done to make sure that the data can not be deduped. See also the dedupratio= parameter.

1.10.6 'port=nnnn': Specify port number for Java sockets.

Vdbench communicates between the master and slave JVMs using Java sockets, using port 5570. If on your system this port is already used by something else you may override this setting using the 'port=nnn' parameter.
Since with Vdbench you can create and execute as many different workloads as you can think of, running more than one Vdbench run at the same time should not be necessary. However, if you choose to do so your Vdbench could run into connection problems if it tries to connect to the same port that another Vdbench execution already is using. To eliminate this risk, Vdbench will increment the port number by one when it gets a connection failure to see if that connection will be successful. Vdbench will do this a maximum of eight times before it gives up. The end result is that it will try to use ports 5570 through 5578 (default).
Starting Vdbench50402 however the connection port will be released as soon as the master and all of its slaves are connected making the port available again for other copies of Vdbench. The chance that you will ever need more than 8 ports therefore is very small.

You can also use the '-p nnnn' execution parameter to override the port number, or override the used port numbers permanently using the 'portnumbers.txt' file.

1.10.7 'create_anchors=yes': Create anchor parent directory

The anchor directory for a File System Definition (FSD) is automatically created if it does not exist. However, its parent directories will NOT be created. Use 'create_anchors=yes' to also include the creation of the parent directories.

1.10.8 'report=': Generate extra SD reports.

By default Vdbench will create a separate report with detailed statistics for each SD. Starting Vdbench 5.02 Vdbench will no longer create SD reports for each slave or host (200 SDs over 8 slaves plus one host creates 1800 reports!). To still create these detail reports specify report=host_detail or report=slave_detail or abbreviated report=(host,slave) depending on your needs.
report=no_sd_detail will completely suppress the creation of SD reports.

1.10.9 'histogram=': set bucket count and bucket size for response time histograms.

Response time histograms allow you to get a more detailed understanding of I/O response times. Vdbench already reports averages and maximums, but at times it can be very useful to know what variation there is in response times. The histograms for instance can give you some indication as to how much of the I/O was handled from storage system cache and which ones had to really come from spinning disk. Note of course that the response time can vary depending on the data transfer size and the queue depth.

The 'histogram=(default,nn,…,nn)' parameter defines how many histogram buckets there are and what the range of the buckets is. The default is:
histogram=(default,20,40,60,80,100,200,400,600,800,1m,2m,4m,6m,8m,10m,20m,40m,60m,80m,100m,200m,400m,600m,800m,1s,2s) (All in one line).
Values are in microseconds, you may also specify u(micro), m(milli) or s(seconds). You may specify a maximum of 64 buckets).

You may permanently override the default by creating file 'histogram.txt' with a valid histogram parameter inside of your Vdbench installation directory.

For raw I/O workloads (SD/WD), histograms will include all reporting intervals, including warmup. For File System workloads (FSD/FWD) the warmup interval(s) will be excluded.

1.10.10 'formatxfersize=nnnn'

When Vdbench creates new or expands existing disk files specified using SD parameters the default is xfersize=128k. (xfersize=512 for files smaller than 128k). To override this default, specify formatxfersize=nnn.)

1.10.11 'monitor=', External control of Vdbench termination

Vdbench runs terminate after 'elapsed=nnn' seconds, or, when 'seekpct=eof' is used after 'elapsed=nnn' seconds or when EOF is reached on the last SD, whichever is first.

There are TWO ways to allow external control of Vdbench shutdown:

1.10.11.1 Shutdown via temporary file

At the end of each reporting interval Vdbench looks for file '*$temp*/vdbench.shutdown.*$pid*', where '$temp' is the system TEMP directory, and '$pid' is the process-id of the Vdbench master JVM. When that file is found, Vdbench will shut down cleanly.
The '$temp' directory name as used by Java is displayed at the top in logfile.html, look for 'java.io.tmpdir'.

1.10.11.2 Shutdown via monitor= parameter

The monitor=*/monitor/file/name* parameter allows you to terminate the currently active run, or even the complete Vdbench execution.

Vdbench at the end of each reporting interval will look at the monitor file name, and depending on its contents will either terminate the current run, or will terminate Vdbench.

Vdbench will always start with erasing the monitor file, but after that will check the file contents each interval:

- *end_rd* will terminate the current run AFTER the next reporting interval.
- *end_vdbench* will terminate Vdbench AFTER the next reporting interval, skipping any still remaining Run Definitions..

1.10.12 'messagescan=': suppress /var/xxx/messages scan

By default on Solaris /var/adm/messages, and on Linux /var/log/message is scanned every five seconds to look for error messages that possibly could be related to the workload you are running.
The 'messagescan=' parameter give you some control over this.

messagescan=no	suppresses the scan
messagescan=yes	Keeps the default
messagescan=nodisplay	Does the scan, but does not display the messages found on stdout.
messagescan=1000	Stop scanning after 1000 messages. (default)

- messagescan=no suppresses the scan
- messagescan=yes Keeps the default
- messagescan=nodisplay Does the scan, but does not display the messages found on stdout.

1.11 Replay Group (RG) parameter detail

rg=name,	Unique name for this Replay Group (RG).
devices=(xxx,yyy,….)	The device numbers from Swat's flatfile.bin.gz to be replayed.

A Replay Group is a group of devices obtained from Swat whose I/O workload must be replayed on one of more SDs. Vdbench will obtain the maximum lba used for each device from flatfile.bin.gz, and will place them on the target luns, straddling luns if needed.

Example:
rg=group1,devices=(89465200,6568108,110)
rg=group2,devices=(200,300)
sd=sd1,lun=/dev/rdsk/cxtxdxsx,replay=group1
sd=sd2,lun=/dev/rdsk/cytydysy,replay=group1
sd=sd3,lun=/dev/rdsk/cztzdzsz,replay=group2

These Replay Groups make Vdbench work like a primitive volume manager.

A Vdbench replay parameter file can also be created by Swat Trace Facility (STF using the 'File' 'Create replay parameter file' menu option.

1.12 Host Definition parameter detail

1.12.1 'hd=host_label'

The host label is used for cross-referencing and reporting.

Example: hd=localhost or hd=systemA
Use 'localhost' when you want to set values for the *current* host, the host where the Vdbench master JVM is running.
hd=default specifies the default settings to be used for all later HD parameters.

For a complete example, see example 5 below.

1.12.2 'system=system_name'

When running Vdbench in multi-host mode you specify here the system name of the system's IP address, e.g. system=x.y.z.com or system=12.34.56.78.

1.12.3 'jvms=nnn'

This parameter tells Vdbench how many JVMs to use on this host. See 'Multi JVM Execution.

1.12.4 'vdbench=/vdbench/dir/name'

This tells Vdbench where it can find its installation directory on a remote host. Default: the same directory as currently used. Use double quotes (") when a directory name has embedded blanks, for instance on windows systems.

1.12.5 'shell=rsh | ssh | vdbench'

For multi-host execution Vdbench by default uses RSH. You can optionally use SSH. If your target system does not have RSH or SSH or if you can't get the proper settings on the local or remote systems to get RSH or SSH to work you can use Vdbench's own RSH daemon by specifying shell=vdbench. On the target remote system you must do a one time start of './vdbench rsh' to start the Vdbench RSH daemon. This daemon of course will only work with Vdbench, and it is simply a small program that uses Java sockets to start a command and receive stdout/stderr output back. See also 'portnumbers.txt'.

Note: If you have the MKS toolkit installed on your Windows system you may want to remove the MKS version of RSH. Experience has shown that the stdout and stderr output streams created

by this version of RSH do not close properly therefore preventing Vdbench from recognizing the completion of a remote copy of Vdbench.

1.12.6 'user=xxxx'

Used for RSH and SSH. Note: it is the user's responsibility to properly define the RSH or SSH security settings on the local and on the remote hosts.

1.12.7 'mount=xxx'

This parameter is mainly useful when doing some serious multi-host file system testing. If you for instance have 20 target clients that you have connected all to the same file system it is nice not to have to manually create all these mount points and issue the mount commands.

The 'mount=' parameter can be used in two places:

- As part of a Host Definition (HD).
- As part of a Run Definition (RD).

When used as a Host Definition parameter you specify the complete mount command that you want issued on the remote system, e.g.:
mount="mount -o forcedirectio /dev/dsk/c2t6d0s0 /export/h01"

Vdbench will create the mount point directory if needed, in this example /export/h01 and then will issue the mount command.

When used as a Run Definition parameter (RD), you only specify the mount options, e.g. mount="-o noforcedirectio".
Vdbench will replace the (possible) mount options as specified as part of the Host Definition with the newly specified mount options.
When you code 'mount=reset', the original mount command as specified will be executed.

Note, that for normal file system testing operations, each host will need his own FSD parameter, unless the 'shared=yes' FSD parameter is used in which case all hosts can use the same.

1.13 Storage Definition parameter detail

1.13.1 'sd=name': Storage Definition Name

'sd=' uniquely identifies each Storage Definition. The SD name is used by the Workload Definition (WD) and Run Definition (RD) parameters to identify which SDs to use for its workload.
When you specify 'default' as the SD name, the values entered will be used as default for all SD parameters that follow.
The 'name' can contain any free-format name, special characters not allowed.

1.13.2 'lun=lun_name': LUN or File Name

'lun=' describes the name of the raw disk or the file name of the file system file to use. Be careful that you do not specify any disk that contains data that you do not want to lose. Been there, done that ☺. This is the main reason why Vdbench does not require root access. You don't want to lose your root disk.
Please don't think, "I'll only be reading the customer's currently active production disks". One accidental rdpct= with a value different than 100 and all data will be gone. That has happened at least twice!

lun=/dev/rdsk/cxt0d0s0	Name of raw Solaris disk to be used for this SD.
lun=/dev/vx/rdsk/cxt0d0s0	Name of a raw VXVM volume
lun=/home/dir/filename	Solaris file system filename. No control over physical I/O guaranteed since the file system may use system cache.
lun=\\.\d:	Name of raw mounted Windows disk to be used for this SD.
lun=\\.\PhysicalDrive1	Physical number of raw Windows disk to be used for this SD.
lun=c:\temp\filename	Windows: file system file name.
lun=\\.\c:\temp\filename	Windows: raw access to file.

The raw disk or the file system file will be opened for input only unless 'rdpct= is specified with any value other than 100 (default). This allows Vdbench to execute with read-only access to disks or files.
Block zero on a raw volume will never be accessed, this to prevent the volume label from being overwritten. This is accomplished by never allowing Vdbench to generate a seek address of zero, no matter how large the data transfer size is. This also implies that block zero will never be read.

Note: some volume managers will return a NULL block each time that a block is read that has never been written. Though of course this gives wonderful performance numbers the disk is never read and this therefore is NOT a valid workload. When you have a storage device like this, make sure that you first pre-format the whole volume using:

sd=sd1,lun=xx
wd=wd1,sd=*,xf=1m,seekpct=eof,rdpct=0
rd=rd1,wd=*,iorate=max,elapsed=100h,interval=60
(The elapsed time must be large enough to completely format the volume. Vdbench will stop when done, but if the elapsed time is too short Vdbench may terminate too early before the volume has been completely formatted).

You may also specify formatsds=yes as a General Parameter, but remember, this will cause the format to be done each time the same parameter file is used.

Note: When an SD is recognized to be a raw device (lun name starts with "\\" on Windows or "/dev" on Unix), Vdbench will refuse to write to block zero, this to avoid overwriting a volume label.

1.13.3 'host=name'

This parameter is only needed when you do a multi-host run where the lun names on each host are different. For instance if a lun is /dev/rdsk/a on hosta but it is named /dev/rdsk/b on hostb then you'll have to tell Vdbench about it.

The 'lun' and 'host' parameters in this case have to be entered in pairs, connecting a lun name to a host name, e.g.:
sd=sd1,lun=/dev/rdsk/a,host=hosta,lun=/dev/rdsk/b,host=hostb

By default Vdbench assumes that the lun names on each host are identical.

Note: when using SD concatenation on multiple hosts Swat will do an extra verification to make sure you have specified the proper device names, and will even correct your definition if possible. Example:

sd=sd1,host=systemA,lun=/dev/rdsk/1,host=systemB,lun=/dev/rdsk/2
sd=sd2,host=systemA,lun=/dev/rdsk/2,host=systemB,lun=/dev/rdsk/1

Though technically it is of course (sadly) possible that this is correct, it likely is not. Vdbench, by writing 'markers' to these disks on the Vdbench master and then verifying them on the clients will verify that this is incorrect, and will internally change this to:

sd=sd1,host=systemA,lun=/dev/rdsk/1,host=systemB,lun=/dev/rdsk/1
sd=sd2,host=systemA,lun=/dev/rdsk/2,host=systemB,lun=/dev/rdsk/2

(If the disk names are identical on all sides, then the extra 'host=' parameter is not needed)

1.13.4 'count=(nn,mm)'

This parameter allows you to quickly create a sequence of SDs, e.g.
sd=sd,lun=/dir/file,count=(0,8) results in sd0-sd7 for /dir/file0-7.

You may also specify a 'printf' mask, e.g.
sd=sd,lun=/dir/file%04d,count=(0,2) This will result in sd1+2 for /dir/file0001 and file0002.

Note: I have been asked numerous times to also support something like lun=/dev/rdsk/c0*.
I consider this far too dangerous though and therefore decided against it. Once upon a time about 40 years ago I accidentally erased all disk drives on a production system (*05:00 am*). You don't ever want to do that ☺.

1.13.5 'size=nn: Size of LUN or File

'size=' describes the size of the raw disk or file. You can enter this in bytes, kilobytes, megabytes, gigabytes or terabytes (k/m/g/t). If not specified, the size will be taken from the raw disk or from the file. Vdbench supports addresses larger than 2GB.
If this is a non-existing file, or an existing file that is not large enough, a separate Vdbench Run Definition named 'File_format_or_append_for_sd=' is automatically inserted and executed that will do a sequential write or append to the file until the file is full. This replaces the need to create a new file using *mkfile* or other utility.
Use 'formatsds=no' to suppress this auto-create (this is General parameter).

Note: to prevent accidentally creating a huge file in the /dev/rdsk/ directory because an incorrectly entered 50+ digit random hexadecimal lun name, Vdbench will refuse to create new file names that start with /dev/. Bad things happen when your root directory fills up ☺.

You can also use size= to give Vdbench access to only a portion of your volume, though for that you can also use 'range=' below.

1.13.6 'range=(min,max)': Limit Seek Range

By default, the whole SD will be used. To limit the seek range for a workload, specify the starting and ending range of the SD: 'range=(40,60)' will limit I/O activity starting at 40% into the SD and ending at 60% into the SD.
If the max value is larger than 100 but smaller than 200, Vdbench will consider this a wrap across the end of your volume. For instance with range=(90,110) , Vdbench will generate an I/O workload using the last 10% and the first 10% of your volume.

When the values are greater than 200, the values will be considered given in *bytes* instead of in *percentages*; e.g., 'range=(1g,2g)'.

Note: the 'range=' parameter may not be used during Sd concatenation.
Note: hitarea if needed will be at the beginning of the range.

1.13.7 'threads=nn': Maximum Number of Concurrent outstanding I/Os

'threads=nn' specifies the maximum number of concurrent I/O that can be outstanding for this SD. Be aware that depending on the storage subsystem, some of these I/Os may be queued inside of the operating system (wait queue). (On Solaris, check file kstat.html).

Warning: If you specify a LUN that physically consists of multiple disk drives, the thread count determines the maximum concurrency for the LUN, not for the disks. A total of 8 concurrent threads for a total of 16 physical disks will not allow for much concurrency. Also, for Solaris, make sure that your *sd_max_throttle* parameter in */etc/system* allows the requested amount of concurrency.

Note: be aware that the maximum concurrency will only occur when there is enough demand. Requesting 10 iops against a device that can handle 1000 iops will not give you the concurrency that you request with the thread= parameter.

Note: This SD parameter will be ignored when using SD concatenation.

This parameter may be overridden using the 'forthreads=' parameter.

1.13.8 'hitarea=nn': Storage Size for Cache Hits

See read hit percentage for an explanation. Default value is 1MB.

1.13.9 'journal=name': Directory Name for Journal File

Used with journaling only. Journal files are needed for each Storage Definition and are created by default in the current directory. The file names are 'sdname.jnl' and 'sdname.map', where 'sdname' is the name of the SD. The .jnl file is the actual journal file, while the .map file is a backup of the copy of the latest Data Validation map residing in the .jnl file.
See Data Validation and Journaling for a more detailed description of data validation and journaling.

When as part of the Data Validation and journaling testing you bring down your OS it is imperative that all writes to the journal file are synchronous. If your OS or file system does not handle this properly you could end up with a corrupted journal file. A corrupted journal file means that the results will be unpredictable during journal recovery.

Journaling therefore allows you to specify a RAW device, e.g. journal=/dev/xxxx, bypassing the possibly faulty file system code.

1.13.10 'offset=': Don't start at byte zero of a LUN

Vdbench always starts at the beginning of a LUN, but some times it is needed to modify that. Some times a LUN does not start at an exact required physical boundary and this parameter allows you do adjust for that. The offset is in bytes and must be a multiple of 512.

Note: Vdbench never accesses block zero on any raw volume. This has been done to make sure that it never overwrites a volume label and/or vtoc.

1.13.11 'align=': Determine lba boundary for random seeks.

Whenever Vdbench generates an random lba (logical byte address) it by default is always on a block boundary (xfersize=). Use the 'align=' parameter to change that to always generate an LBA on a different alignment. The align= value is in bytes and must be a multiple of 512. This parameter may not be used when dedupratio= is specified.

1.13.12 'openflags=': control over open and close of luns or files

This parameter allows you to control what parameters are passed to the system's open and close functions. By default write operations are handled according to how the file system is mounted or for raw devices, how the device normally operates. This can mean that a write operation completes as soon as the data is stored in system cache. This makes for very good performance, but does not really exercise the storage.

Openflags can be specified for SD, WD, FSD, FWD, and RD parameters.

Options (you can create any combination of these)

Solaris:	(xx_SYNC descriptions found in man open) Please believe me, I have never really figured out what these options *mean*.
o_dsync	Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion
o_rsync	Read I/O operations on the file descriptor complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in oflag, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in oflag, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.
o_sync	Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.
0x……	Any hex value, to be passed to the open() function.
fsync	Call fsync() before the file is closed.
directio	Calls the directio() function after the file is opened, using 'DIRECTIO_ON'
directio_off	Calls the directio() function after the file is opened, using 'DIRECTIO_OFF'. This one is meant to be used if a previous failed I/O run left the target file name with directio active and you want to forcibly remove that status.
clear_cache	When using directio() for an NFS mounted file, any data still residing in file system cache will continue to be used, circumventing the directio() request. This option will forcibly clear any existing data from cache using mmap() functions.

Linux:	
o_direct or directio	Gives you raw access to a full volume (no partition). This parameter is *required* when using /dev/xxx volumes
o_dsync o_rsync o_sync	These three all pass '0x01000' to the Linux open() function. *I highly suggest you check /usr/include/bits/fcntl.h since not all flavors of Linux use the same bits. Then if needed code openflags=0x…. instead.*

fsync	Call fsync() before the file is closed.
0x……	Any hex value, to be passed to the open() function.

Windows:	
directio	Opens a file using the ' FILE_FLAG_NO_BUFFERING' flag

AIX	
o_dsync	Passes 0x00400000 to open().
o_rsync	Passes 0x00200000 to open().
o_sync	Passes 0x00000010 to open().
o_direct or directio	Passes 0x08000000 to open().
0x……	Any hex value, to be passed to the open() function.

MAC OSX:	
OSX does not allow flags to be passed to open(), instead, after the open() request Vdbench calls the fcntl() function.	
f_nocache	Decimal 48 is passed to fcntl()
directio	Decimal 48 is passed to fcntl()
nnn	Decimal nnn is passed to fcntl()

1.13.13 streams=: Independent sequential streams.

Note: In Vdbench 503 this was an SD parameter; it has been moved to WD, and its meaning has slightly changed (the 'stream size' subparameter has been removed).

The streams=nn parameter overrides the default processing done when reading or writing sequential data. By default Vdbench just reads or writes the whole lun or file (SD, or concatenated SD), or only a portion of it when the range= parameter is used, but the streams= parameter allows multiple concurrent sequential streams to be active.

streams=*stream_count.* This parameter works together with the threads= parameter. The file or lun is split into 'stream_count' pieces.

Example: wd=wd1,…..,streams=10

Each SD (or the concatenated SD) is split into 10 equally sized smaller pieces. The requested threads are spread out over the streams, with as a possible result that some streams may not get the same amount of threads than other streams. This is unacceptable, so Vdbench will increase the threadcount to make sure each stream gets the same amount of threads.
Any adjustment in the threadcount will be reported to you.

Note that, since this is sequential I/O, a stream may run on only ONE JVM, this to avoid for instances reading blocks 1,1,2,2,3,3,4,4, etc.

Note: The stream count must be a multiple of the amount of JVMs used, this to make sure that the requested stream count can be equally spread around these JVMs. This also will be adjusted by Vdbench if needed.
See Multi JVM execution.

Note: rhpct= and whpct= are ignored when streams are requested. However, if you really want to run cache hits with streams, just code size='small' to force all I/O to cache.

Note: Only ONE Workload Definition (WD) may be active when using streams.

1.14 Workload Definition parameter detail

The Workload Definition parameters describe what kind of workload must be executed using the storage definitions entered. Note that a lot of these parameters can be overridden within a Run definition (RD) using ‘forxxx=’ parameters.

Example: wd=wd1,sd=(sd1,sd2),rdpct=100,xfersize=4k

wd=default	Sets defaults for all WDs that are entered later.

wd=name	Unique name for this Workload Definition (WD)
host=host_label	Specify here which host you want this to run on.
iorate=nn	Workload specific I/O rate
openflags=	See openflags=
priority=	Workload specific I/O priority.
range=(nn,nn)	Limit seek range to a defined range within an SD.
rdpct=nn	Read percentage. Default 100.
rhpct=nn	Read hit percentage. Default 0.
sd=xx	Name(s) of Storage Definition(s) to use
seekpct=nn	Percentage of random seeks. Default seekpct=100 or seekpct=random.
skew=nn	Percentage of skew that this workload receives from the total I/O rate.
stride=(min,max)	To allow for skip-sequential I/O.
whpct=nn	Write hit percentage. Default 0.
xfersize=nn	Data transfer size. Default 4k.
xfersize=(nn,%%,…)	Data transfer size distribution.
xfersize=(min,max,align)	Generate xfersize as a random value between min and max.

1.14.1 'wd=name': Workload Definition Name

'wd=name' uniquely identifies each Workload Definition. The WD name is used by the Run Definition parameters to identify which workloads to execute. When you specify ‘default’ as the WD name, the values entered will be used as default for all WD parameters that follow.

1.14.2 ‘host=host_label’

This parameter is only needed for multi-host runs where you do not want each workload to run on each host. For example:
wd=wd1,host=hosta,….
wd=wd2,host=hostb,…

1.14.3 'sd=name': SD names used in Workload

'sd=' selects a specific SD for this workload. A single SD name can be specified as 'sd=sd1', multiple SDs can be specified as either 'sd=(sd1,sd2,sd3,..)', a range as in ‘sd=(sd1-sd99)’, using a wildcard character 'sd=sd*', or a combination of these.
Note: When using an SD range, leading zeros are not allowed, e.g. (sd01-sd09).

You may specify SDs also as an RD= subparameter.

1.14.4 'rdpct=nn': Read Percentage

'rdpct=' specifies the read percentage of the workload. rdpct=100 means 100% read; rdpct=0 means 100% write. rdpct=80 means a read/write ratio of 4:1, etc.

The default is 100% read. If there are no workloads for a specific SD with a read percentage other than 100 this SD is opened for input only.

This parameter may be overridden using the 'forrdpct=' parameter.

1.14.5 'rhpct=nn' and 'whpct=nn': Read and Write Hit Percentage

'rhpct=' and 'whpct=' specify the cache hit percentage that Vdbench will attempt to generate. This parameter is only useful when accessing raw devices or file systems mounted with 'forcedirectio' (or using 'openflags'). For this to work, each volume is divided into two parts: the first one-megabyte of storage will be accessed for cache hits and is called the ***hit area***. The remaining space on the SD will be accessed for cache misses and is called the ***miss area***.

When Vdbench needs to generate a cache hit, it generates an I/O to the hit area, assuming that the data accessed is, or soon will be, residing in cache. Cache misses will be targeted toward the miss area, assuming that the miss area is large enough to ensure that most random accesses are cache misses. As you can see, this will only be useful when Vdbench has control over which physical volume and physical blocks will be ultimately read or written (so no file system cache).

The **'hitarea=nn'** and **'forhitarea='** parameters have been created to allow control over a volume's cache working set size. Some cached storage subsystems have different performance characteristics if too small a subset of the available cache is used. The total size of the SD hit or miss area must be at least 4 times the largest xfersize used, but Data Validation has much more stringent requirements, enough blocks to satistfy at least 2000 times the largest xfersize used.

These parameters may be overridden using the 'forrhpct=' or 'forwhpct=' parameters.

1.14.6 'xfersize=nn': Data Transfer Size

'xfersize=' specifies how much data is transferred for each I/O operation; allows (k)ilo and (m)ega bytes. This parameter may be overridden using the 'forxfersize' parameter.

Example: xfersize=4k (default)

You may also specify a distribution of data transfer sizes. Specify pairs of transfer size and percentages; the total of the percentages must add up to 100.
Example: xfersize=(4k,10,8k,10,16k,80)

A third option uses three values: xfersize=(min,max,align). This causes a random value between min and max, with a multiple of align to be generated. This also requires the use of the SD align= parameter. This parameter may not be used when dedupratio= is specified.

1.14.7 'skew=nn': Percentage skew

'skew=' specifies the percentage of the run's total I/O rate that will be generated for this workload. By default the total I/O rate will be evenly divided among all workloads. However, if the skew value is nonzero, a percentage of the requested I/O rate equal to the percentage skew value will be apportioned to one workload, with the remaining skew evenly divided among the workloads that have no skew percentage specified. The total skew for all workloads used in a Run Definition must equal 100%.

Example: Five workload definitions, one workload specifies 'skew=60'. This workload receives 60 percent of the requested I/O activity while the other four workloads each receive 10% of the requested I/O rate, with a total of 100%.

Vdbench generates I/O workloads by sending new I/O requests to a volume's (SD) internal work queue. This work queue has a maximum queue depth of 2000 per SD. If an SD cannot keep up with its requested workload and the queue fills up, Vdbench will not be able to generate new I/O requests for this and all other SDs until space in the queue becomes available again. This means that if you send 1000 IOPS to an SD that can handle only 100 IOPS, and 50 IOPS to a similar device, the queue for the first device will fill up, and I/O request generation for the second device will be held up. This has been done to enable Vdbench to preserve the requested workload skew while still allowing for a temporary 'backlog' of requested I/Os.

Note: see also the ' abort_failed_skew=nn' parameter.

To accommodate users that still would like to run an ***uncontrolled*** workload see 'iorate=max'.

1.14.8 'seekpct=nn': Percentage of Random Seeks

'seekpct=' specifies how often a seek to a random lba will be generated. See also the stride= parameter for skip sequential processing.

seekpct=100 or seekpct=random	Every I/O will go to a different random seek address.
seekpct=0 or seekpct=sequential	There will be NO random seeks, and the run will therefore be purely sequential. When the end of the volume or file is reached, Vdbench will continue at the beginning, unless 'seekpct=eof' (see below) is specified. Be aware that if the volume size is smaller than the cache size, continued processing will be all cache hits.

	Note: a 100% sequential workload is allowed to run on only ONE JVM, because without that limitation we could end up reading blocks 1,1,1,2,2,2,3,3,3,etc. Wonderful marketing performance numbers, but we're trying to stay honest here.
seekpct=20	On average, 20% of the I/O operations will start at a new random seek address. This means that on average there will be one random seek, with 5 consecutive blocks of data transferred. See stride= below for skip sequential I/O.
seekpct=-1 or seekpct=eof	A negative *one* value causes a sequential workload to terminate as soon as end-of-file is reached. Vdbench will continue until all workloads using seekpct=eof have reached EOF.

The randomizer used to generate a seek address is seeded using the host's time of day in micro seconds multiplied by the relative position of the Storage Definition (SD) defined for the workload (WD). For sequential processing, I/O for raw devices always starts at the ***second*** block as defined by the data transfer size (block zero is never used).

This parameter may be overridden using the 'forseekpct=' parameter, though forseek will only accept numeric values, not 'random' or 'seq'.

1.14.9 stride=(min,max): Skip-sequential I/O.

The stride= parameter changes the behavior when a new lba has to be generated because of the use of the seekpct= parameter. Instead of generating a brand new random lba, I/O skips 'n' blocks, 'n' being a random value between min and max, with a multiple of the current selected data transfer size. When end of volume is reached we start again at the beginning.

1.14.10 'range=nn': Limit Seek Range

By default, the whole SD will be used. To limit the seek range for a workload, specify the starting and ending range of the SD: 'range=(40,60)' will limit I/O activity starting at 40% into the SD and ending at 60% into the SD.
If the max value is larger than 100 but smaller than 200, Vdbench will consider this a wrap across the end of your volume. For instance with range=(90,110) , Vdbench will generate an I/O workload using the last 10% and the first 10% of your volume.

When the values are greater than 200, the values will be considered given in *bytes* instead of in *percentages*; e.g., 'range=(1g,2g)'.

Also see Hot Bands.

1.14.11 'iorate=' Workload specific I/O rate.

Normally the I/O rate for a workload is controlled by the rd=xxx,iorate= parameter, together with the workload skew= parameter. With the workload specific iorate= parameter you can now give a FIXED I/O rate to a workload, while the other workloads continue to be controlled by the rd=xxx,iorate= and the workload skew parameters.
When used, 'priority=' must also be specified..

This option was initially created to test a 'what if': "If I run a Video On Demand (VOD) workload , what will the impact on performance be if I add some maintenance or video editing workload?".

1.14.12 'priority=' Workload specific I/O priority.

All I/O in Vdbench is scheduled using the expected I/O arrival times, which is obtained from the requested I/O rate. For instance, iorate=100 starts a new I/O on average every 1000/100 = 10 milliseconds. There are no I/O priorities within devices or workloads.
The new 'priority=' parameter attempts to change this.
Normally I/O in Vdbench is handled using internal 'work' fifo queues. When the priority= parameter is used, any work in a higher priority fifo queue is processed before any lower priority fifo is checked causing I/O for that higher priority workload to be started first.
When any priority is specified, ALL workloads will have to have a priority specified. Priorities go from 1 (highest) to 'n' (lowest), and must be in sequence (priorities must be for instance 1 and 2, not 1 and 3).
Warning though: allowing for a specific workload iorate and priority does not guarantee that if you ask for 100 IOPS and your system can do only 50 that Vdbench magically gets the workload done ☺.

1.15 Run Definition for raw I/O parameter detail

For parameters specific to file system testing, see RD parameters for file system testing, detail.

The Run Definition parameters specify which of the earlier defined workloads need to be executed, which I/O rates need to be generated, and how long to run the workloads. One Run Definition can result in multiple actual runs, depending on the parameters used.

Example: rd=run1,wd=(wd1,wd2),iorate=1000,elapsed=60,interval=5

rd=default	Sets defaults for all RDs that are entered later.
rd=name	Unique name for this Run Definition (RD).
wd=xx	Workload Definitions to use for this run.
sd=xxx	Which SDs to use for this run (Optional).
curve=(nn,nn,..)	Data points to generate when creating a performance curve.
stopcurve=n.n	Stop running curve datapoints when response time > n.n ms.
distribution=(x[,variable]	I/O inter arrival time calculations: exponential, uniform, or deterministic. Default exponential.
elapsed=nn	Elapsed time for this run in seconds. Default 30 seconds.
maxdata=nnn	Stop the run after nnn bytes have been read or written, e.g. maxdata=200g. Vdbench will stop at the lower of elapsed= and maxdata=.
endcmd=cmd	Execute command or script at the end of the last run
(for)compratio=nn	Multiple runs for each compression percentage.
(for)hitarea=nn	Multiple runs for each hit area size.
(for)hpct=nn	Multiple runs for each read hit percentage.
(for)rdpct=nn	Multiple runs for each read percentage.
(for)seekpct=nn	Multiple runs for each seek percentage.
(for)threads=nn	Multiple runs for each read thread count.
(for)whpct=nn	Multiple runs for each write hit percentage.
(for)xfersize=nn	Multiple runs for each data transfer size.
Most forxxx parameters can just be abbreviated to their regular name, e.g. xfersize=(..,..)	
interval=nn	Reporting interval in seconds. Default 'min(elapsed/2,60)'
iorate=(nn,nn,nn,…)	One or more I/O rates.
iorate=curve	Create a performance curve.
iorate=max	Run an ***uncontrolled*** workload.
iorate=(nn,ss,nn,ss,…)	nn,ss: pairs of I/O rates and seconds of duration for this I/O rate. See also 'distribution=variable'.
mount=xxx	See HD mount parameter
openflags=xxxx	Pass specific flags when opening a lun or file

pause=nn	Sleep 'nn' seconds before starting next run.
replay=(filename,…)	File name used for Swat I/O replay (file 'flatfile.bin' from Swat)
startcmd=cmd	Execute command or script at the beginning of the first run
warmup=nn	Override warmup period.

1.15.1 'rd=name': Run Name

'rd=*name*' defines a unique name for this run. Run names are used in output reports to identify which run is reported.
When you specify 'default' as the RD name, the values entered will be used as default for all SD parameters that follow.

1.15.2 'wd=': Names of Workloads to Run

'wd=' identifies workloads to run. Specify a single workload as 'wd=wd1' or multiple workloads either by entering them individually 'wd=(wd1,wd2,wd3)', a range 'wd=(wd1-wd3)' or by using a wildcard character: 'wd=wd*'.
The total skew percentage specified for all requested workloads must equal 100.
Note: When using a WD range, leading zeros are not allowed, e.g. (wd01-wd09)

1.15.3 'sd=xxx'

Normally you specify the SDs to be used as part of a Workload Definition (WD) parameter. However, when you specify all workload parameters using the available 'forxx' RD options, the WD parameter really is not necessary at all, so you can now specify the SD parameters during the Run Definition.
You can specify both the WD and the SD parameters though. In that case the SD parameters specified here will override the SD parameters used when you defined the WD= workload. If you do not specify the WD parameter, all defaults as currently set for the Workload Definition are used.

1.15.4 'iorate=nn': One or More I/O rates

iorate=100	Run a workload of 100 I/Os per second
iorate=(100,200,…)	Run a workload of 100 I/Os per second, then 200, etc.
iorate=(100-1000,100)	Run workloads with I/O rates from 100 to 1000, incremented by 100.
iorate=curve	Run a performance curve. See below.
iorate=max	Run the maximum **uncontrolled** I/O rate possible. See below.

iorate=(nn,ss,nn,ss,…)	nn,ss: pairs of I/O rates and seconds of duration for this I/O rate. See also 'distribution=variable'.

'iorate=curve': Run a performance curve.

- When no skew is requested for any workload involved, determine the maximum possible I/O rate by running 'iorate=max'. After this 'max' run, the target skew for the requested workloads will be set to the skew that was observed. Workloads of 10%, 50%, 70%, 80%, 90% and 100% of the observed maximum I/O rate and skew will be run. Target I/O rates above 100 will be rounded up to 100; I/O rates below 100 will be rounded up to 10. Curve percentages can be overridden using the 'curve=' parameter.
- When any skew is requested, do the same, but ***within*** the requested skew.
- stopcurve=n.n will prevent the next curve datapoints from being run once the average response time reaches n.n milliseconds.

'iorate=max': Run the maximum I/O rate possible.

- When **no** skew is requested, allow each workload to run as fast as possible without controlling skew. This is called an **uncontrolled** max workload.
- When **any** skew is requested, allow all workloads to run as fast as possible ***within*** the requested skew.

During a Vdbench replay run, the I/O rate by default is set to the I/O rate as it is observed in the trace input. This I/O rate is reported on the console and in file *logfile.html*.
If a different I/O rate is requested, the inter-arrival times of all the I/Os found in the trace will be adjusted to allow for the requested change. Note though that a one hour trace replayed at twice the iops will last only 30 minutes.

1.15.5 'curve=nn': Define Data points for Curve

The default data points for a performance curve are 10, 50, 70,80, 90, and 100%. To change this, use the curve= parameter. Example: curve=(10-100,10) creates one data point for each 10% of the maximum I/O rate. Target I/O rates above 100 will be rounded up to 100; I/O rates below 100 will be rounded up to 10.
Remember: each data point takes 'elapsed=nn' seconds!

1.15.6 'elapsed=nn: Elapsed Time

This parameter specifies the elapsed time in seconds for each run. This value needs to be at least twice the value of the reporting interval below. Each requested workload runs for 'elapsed=' seconds while detailed performance interval statistics are reported every 'interval=' seconds. At the end of a run, a total is reported for all intervals *except* for the first interval (or warmup=nn

duration). The requirement to have at least 2 intervals allows us to have at least 1 reporting interval included in the workload total.
See also: maxdata and 'format=limited'.

Note that if you specify seek=eof your run will stop at the earlier of reaching EOF, or the elapsed time. This means that a seek=eof run for a 10TB lun with elapsed=10 will stop after 10 seconds! In that case, just specify elapsed=100h.
The same of course also is valid when using maxdata=x.

1.15.7 'interval=nn': Reporting Interval

This parameter specifies the duration in seconds for each reporting interval. At the end of each reporting interval, all statistics that have been collected are reported.

Note: when doing very long runs for instance over the weekend, the amount of detail data reported can become overwhelming. A more reasonable interval duration may be appropriate, for instance 60 seconds. However, if you have a poorly performing storage device that never seems to be able to get to a stable workload I/O rate you may want to choose the lowest reporting interval possible. The longer your reporting interval, the more you will hide possible performance problems.
See also the report=no_sd_detail parameter if you want to limit the amount of output.

1.15.8 'warmup=nn': Warmup period

By default Vdbench will exclude the first interval from its run totals. The warmup value will cause the first 'warmup/interval' intervals to be excluded. Using this parameter also changes the meaning of the 'elapsed=' parameter to mean 'run elapsed= seconds *after* the warmup period completes'.

1.15.9 'maxdata=': stop after nnn bytes.

maxdata=, maxdata_read=, max_data_written:
Normally a run terminates after elapsed= seconds. maxdata= will terminate the run after the shorter of elapsed= or maxdata= bytes have been read or written.
Note that the byte count starts AFTER warmup= seconds, and that maxdata is only checked at the end of a reporting interval.

This parameter is available both for both raw and file system workloads, with the following difference: for raw workloads, any specified value less than 100 will be used to multiply this value with the total size of all currently used SDs. For instance two SDs, each of 10gb, using maxdata=5 will stop this run after 2*10gb*5=100gb worth of data has been accessed AFTER the warmup period.
Remember though: your elapsed time must be long enough to get this far.

1.15.10 'distribution=xxx': I/O arrival time distribution

distribution=([exponential][uniform][deterministic],[variable],[spike])
distribution=([e][u][d],[variable],[spike])

- Exponential: exponential distribution. Default. In simple terms, an exponential arrival rate is a good distribution to achieve lots of I/O bursts. It is a good method to approximate a large application with many users. An exponential arrival rate is a classic modeling approach to describe a general arrival distribution of independent events. It causes significant queuing to occur even when the device utilization is only 50% busy.
- Uniform: uniform distribution
- Deterministic: deterministic: all I/O is evenly spread out using fixed inter arrival times.
- Variable: changes the meaning of the iorate= parameter, defining a set of pairs, first in an I/O rate, then the amount of seconds to use that I/O rate. This causes I/O to generate a variable I/O rate.
- Spike: the I/O for the above mentioned variable I/O rates will be all started together at the beginning of each second instead of them being spread out using the requested exponential or uniform distribution.

Example:
rd=rd1,…,iorate=(100,10,1000,5,50,10),dist=variable

Result: 100 iops for ten seconds, 1000 iops for five seconds, 50 iops for ten seconds and then again 100 iops. Make sure your elapsed time covers your requested amount of seconds.
The total amount of seconds specified may not be more than 3600 seconds.

1.15.11 'pause=nn': Sleep 'nn' Seconds

When doing multiple runs, the 'pause=nn' parameter causes Vdbench to go to sleep for 'nn' seconds before it starts the next run. This parameter is ignored for the first run. 'pause=nn' can be used to allow a storage controller some time to catch its breath and complete things like emptying cache.

1.15.12 Workload parameter specification in a Run Definition.

The original objective of all the forxxx parameters was to allow a user to override most of all Workload Definition (WD) parameters specified earlier to create complex, varying workloads, like forxfersize=(1k-1m,d).
Once I observed that some users were specifying here frequently just single parameters like forx=16k I realized that this started making the Workload Definition obsolete. Yes, you'll always need multiple WDs to allow the running of *different concurrent* workloads, but for a workload without any forxxx variations WDs were not really needed anymore.

To make life easier for my users I then added the sd= parameter to a Run Definition, and from that point on indeed the WD became obsolete for this type of non-varying run.
The next step then was of course to not even ask you to specify forxfersize=, but just simply xfersize=. Internally of course Vdbench still treats it as a forxxx parameter, but as far as the parameter definition, you can now just specify xfersize=4k.

Previous:
sd=sd1,lun=/dev/xxx
wd=wd1,sd=*,rdpct=100,xfersize=4k
rd=rd1,wd=wd1,iorate=max,elapsed=60,interval=1

New (And of course you can still code xfersize=(4k,8k):
sd=sd1,lun=/dev/xxx
rd=rd1,sd=*,iorate=max,elapsed=60,interval=1,rdpct=100,xfersize=4k

1.15.12.1 'sd=xxx' Specify SDs to use

If you did not specify a Workload Definition you may specify the SDs to be used here.
If you use both the sd= and wd= parameters, this will override the SDs specified in the Workload Definition.

Three little tricks: sd=single, sd=range and sd=setsofN.

- sd=single: Let's say you have 10 devices you want to test. Just code an SD for each of them, sd1-sd10. Instead of having to specify one Run Definition for each, just code sd=single, and Vdbench will repeat the current RD once for each SD.
- sd=range: To do a test for sd1, then sd1-sd2, then sd1-sd3 etc. code sd=range and Vdbench will take care of it.
- sd=setsofN: Vdbench will create a Run Definition for each set of 'N' randomly selected SDs. If you have 512 drives, sd=single just is too slow. Example: sd=setsof4

1.15.12.2 '(for)xfersize=nn': Create '*for*' Loop Using Different Transfer Sizes

The 'forxfersize=' parameter is an override for all workload specific xfersize parameters and allows multiple automatic executions of a workload with different data transfer sizes.

forxfersize=4k	One run with 4k transfer size.
forxfersize=(4k,8k,12k,16k)	One run each for 4k, 8k, etc.
forxfersize=(4k-32k,4k)	One run each from 4k to 32k in increments of 4k.
forxfersize=(1k-128k,d)	One run each from 1k to 128k, each time doubling the transfer size. (1k,2k,4k,8k,etc).
forxfersize=(128k,1k,d)	'Reverse' double: (128k,64k,32k,etc)

See Order of Execution for information on the execution order of this parameter.

1.15.12.3 '(for)threads=nn': Create *'for'* Loop Using Different Thread Counts

The 'forthreads=' parameter is an override for all storage definition specific thread parameters and allows multiple automatic executions of a workload with different numbers of threads.

Note: the meaning of this parameter changes when using SD concatenation. During SD concatenation this parameter specified the total amount of threads that will be shared by all storage definitions.

forthreads=32	Do one run with 32 threads for each SD.
forthreads=(1,2,3,4)	Do one run each with 1, 2, 3 and 4 threads.
forthreads=(1-5,1)	Do one run each from 1 to 5 threads in increments of one.
forthreads=(1-64,d)	Do one run each from 1 to 64 threads, each time doubling the thread count.
forthreads=(64-1,d)	'Reverse' double: (64,32,16,etc)

See Order of Execution for information on the execution order of this parameter.

1.15.12.4 '(for)rdpct=nn': Create *'for'* Loop Using Different Read Percentages

The 'forrdpct=' parameter is an override for all workload specific *rdpct* parameters, and allows multiple automatic executions of a workload with different read percentages.

forrdpct=50	Do one run with 50% reads
forrdpct=(10-100,10)	Do one run each with read percentage values ranging from 10 to 100 percent, incrementing the read percentage by 10 percent each time.

See Order of Execution for information on the execution order of this parameter.

1.15.12.5 '(for)rhpct=nn': Create *'for'* Loop Using Different Read Hit Percentages

The 'forrhpct=' parameter is an override for all workload specific *rhpct* parameters, and allows multiple automatic executions of a workload with different read hit percentages.

forrhpct=50	Do one run with 50% read hits
forrhpct=(10-100,10)	Do one run each with read hit percentage values ranging

	from 10 to 100 percent, incrementing the read hit percentage by 10 percent each time.

See Order of Execution for information on the execution order of this parameter.

1.15.12.6 '(for)whpct=nn': Create *'for'* Loop Using Different Write Hit Percentages

The 'forwhpct=' parameter is an override for all workload specific *whpct* parameters, and allows multiple automatic executions of a workload with different write hit percentages.

forwhpct=50	Do one run with 50% write hits
forwhpct=(10-100,10)	Do one run each with write hit percentage values ranging from 10 to 100 percent, incrementing the write hit percentage by 10 percent each time.

See Order of Execution for information on the execution order of this parameter.

1.15.12.7 '(for)seekpct=nn': Create *'for'* Loop Using Different Seek Percentages

The 'forseekpct=' parameter is an override for all workload specific *seekpct* parameters, and allows multiple automatic executions of a workload with different seek percentages.

forseekpct=50	Do one run with 50% seek
forseekpct =(10-100,10)	Do one run each with seek percentage values ranging from 10 to 100 percent, incrementing the seek percentage by 10 percent each time.

See Order of Execution for information on the execution order of this parameter.

1.15.12.8 '(for)hitarea=nn': Create *'for'* Loop Using Different Hit Area Sizes

The 'forhitarea=' parameter is an override for all Storage Definition hit area parameters, and allows multiple automatic executions of a workload with different hit area sizes:

forhitarea=4m	Do one run with 4MB hit area
forhitarea =(10m-100m,10m)	Do one run each with hit area values ranging from 10 MB to 100 MB, incrementing the hit area by 10 MB each time.

See Order of Execution for information on the execution order of this parameter.

1.15.12.9 '(for)compratio=nn': Create *'for'* Loop Using Different compression ratios.

The 'forcompratio=' parameter is an override for the compratio= parameter, and allows multiple automatic executions of a workload with different compression ratios.

1.15.12.10 Order of Execution Using 'for*xxx*' Parameters

The order in which the 'forthreads=', 'forxfersize=', 'forrdpct=', 'forrhpct=' , 'forwhpct=', 'forseekpct=' and 'forhitarea=' parameters are found in the input will determine the order in which the requested workloads will be executed.
Treat this as a sequence of embedded 'for' loops. Using 'forthreads=' and 'forxfersize' as an example:

- for (all threads) { for (all xfersizes) { for (all I/O rates) } } versus
- for (all xfersizes) { for (all threads) { for (all I/O rates) } } depending on what parameter came first.

1.16 Hot banding and SD concatenation:

Vdbench 504 introduces two main new pieces of functionality for raw I/O: "hot band" or "hotband" workloads and SD concatenation.

Note: Hot banding does NOT require SD concatenation, and SD concatenation does NOT require hot banding though it is highly *recommended* to use hot banding when using SD concatenation.

Hotband parameter, e.g. wd=wd1,sd=*,hotband=(10,20)	
hotband=(start,end)	Specify the starting and ending point of a hotband. Can be specified in either percentages 'hotband=(10,20)', or in bytes, e.g. 'hotband=(30g,60g)'

Hot Band workloads: Vdbench until now has basically had three types of workloads when doing raw I/O: pure sequential, pure random, and a mix of random and sequential. Sequential speaks for itself, but it is the 'random' that started becoming a problem. Random for I/O has always meant that for each I/O a new random seek address ***over the whole volume*** (SD) was generated. We all know of course, that it is highly unlikely that there are any real workloads that access data in a pure random fashion. Typically there is some locality of reference going on, where a big portion of the I/O is done against a smaller subset of the amount of storage that is available. Of course, doing only pure random I/O against all available storage also does not make for a very nice test for storage devices that have one or more levels of cache.

To resolve this problem a new type of random workload has been created, a workload called 'hot-banding'. The new “hot band” I/O workload provides a workload that considers the contribution of read caching. This workload allows skewed access across one or more subsets of the available storage. This skewed access tends to hold data in cache and creates "cache hits" for improved throughput and performance.

Introducing this new hot banding workload created a different problem though: until now, any workload given to Vdbench was executed identically on each single volume given to Vdbench. With an objective to create hot bands forcing identical hot bands to every single volume of course is a huge contradiction.

This in turn required the second large change to Vdbench, something that we call "SD concatenation", with SD of course being a Vdbench Storage Definition, any volume/lun/disk/drive/slice/partition/file that you tell Vdbench to use.
SD concatenation basically is a simple volume manager, where all volumes (SDs) given to a workload are treated as if they are one large concatenated storage device. The hot band workloads then are executed against these concatenated volumes.

This of course was not all: normally, by using the 'threads=' parameter you tell Vdbench what the maximum amount of concurrent outstanding I/O may be against each SD. With hot band workloads dedicating threads to an SD that possibly is not or rarely used of course does not make

sense, so, when defining hot bands, the amount of threads to be used will be given to the concatenated SD instead. Be careful though, the default thread count is still just eight, so make sure you give the workloads what they need.

How many threads do you need? If you are running just one single workload it may not be too difficult to figure that out, but if you run a dozen different workloads concurrently things can get ugly fast. So for that, you can, for SD concatenation only, specify a thread count for a Run Definition (RD), and all threads will be shared between all SDs, and all workloads. Note though that you may end up having 500 threads and 500 volumes, but all 500 threads will be used for the slowest volume.

In normal Vdbench operation the Run Definition 'threads=' or 'forthreads=' parameter overrides the threads= value for each individual SD. With SD concatenation active however this is the TOTAL amount of threads that will be shared.

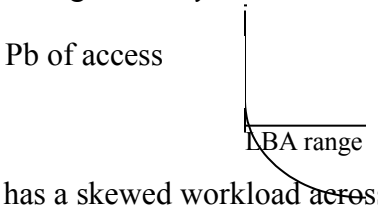
SD concatenation and sequential I/O: Of course, no good deed goes unpunished. When you have twelve drives and treat them as one large concatenation, and are doing sequential I/O on that concatenated SD, you get only ONE active drive. That of course then results in slow (one drive only) throughput. So for this, either don't use concatenation, or ask Vdbench to start doing independent sequential streams using the streams= parameter.

New or changed parameters:

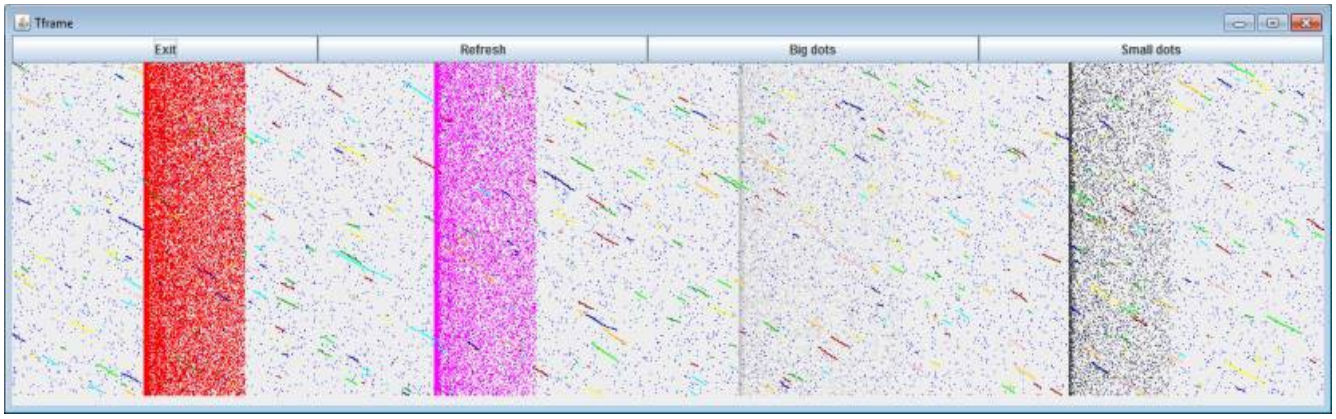
- concatenate=yes Must be specified BEFORE your first SD parameter. Note that you can not have a mix of concatenated and not concatenated SDs.
- wd=xxx,sd=(sd1,sd2,….) With SD concatenation active all these SDs will be treated as ONE large concatenated SD.
- wd=xxx,sd=(…..),……,threads=nn This specifies the total amount of threads to be shared by this workload. Can only be used with SD concatenation.
- rd=xxx,……,threads=nn With SD concatenation active this is the amount of threads shared between all SDs and workloads.
- sd=xxx,……,streams=nn This parameter has been moved and now is a Workload Definition (WD) parameter, e.g. wd=xxx,….,streams=nn.
- sd=xxx,…,streams= This parameter has been moved to the Workload Definition and had its meaning slightly changed.

1.17 'Hotband=(min,max)': Create a skewed workload over a Limited Seek Range

By default, the whole SD or concatenated SD will be used to generate a random LBA. To limit the lba range for a workload, specify the starting and ending range of the SD: 'hotband=(40,60)' will limit I/O activity starting at 40% into the SD and ending at 60% into the SD. If the max value is larger than 100 but smaller than 200, Vdbench will consider this a wrap across the end of your SD. For instance with hotband=(90,110) , Vdbench will generate an I/O workload using the last 10% and the first 10% of your volume. When the values are greater than 200, the values will be considered given in *bytes* instead of in *percentages*; e.g., 'Hotband=(1g,2g)'.



Hotband has a skewed workload across the defined range. The probability (Pb) accessing a block in the range is biased toward lower LBAs It is intended to simulate something similar to database index. This access will allow for systems with a small cache to have a small cache hit. Systems with larger caches, have larger cache hit rates. The cache rate does not increase linearly as system cache size increases. Here is a sample scatter plot of four hotbands and the intensity of access is skewed to lower LBA. The X axis is the storage LBA range, the Y axis is time.



1.18 Data Deduplication:

Data Deduplication is built into Vdbench with the understanding that the dedup logic included in the target storage device looks at each n-byte data block to see if a block with identical content already exists. When there is a match the block no longer needs to be written to storage and a pointer to the already existing block is stored instead.

Since it is possible for dedup and data compression algorithms to be used at the same time, dedup by default generates data patterns that do not compress.

Dedup parameters are usually General Parameters and must be specified at the top of your parameter file BEFORE the first Host Definition (HD) or if HD is not used, before the first SD or FSD. Dedup parameters, if needed, can also be specified as a set of sub parameters for an SD.	
dedupunit=nn	The size of a data block that dedup tries to match with already existing data. There is no longer a default of 128k. There may be only ONE dedupunit parameter in a Vdbench execution, and that must be specified as a General Parameter (though the other Dedup parameters may be specified for each SD).
dedupratio=n	Ratio between the original data and the actually written data, e.g. dedupratio=2 for a 2:1 ratio. Default: no dedup, or dedupratio=1
dedupsets=nn	How many different sets or groups of duplicate blocks to have. See below. Default: dedupsets=5% (You can also just code a numeric value, e.g. dedupsets=100)
deduphotsets=(n,n, . . .)	How many hot sets, in pairs, e.g. deduphotsets=(10,2,20,5) .Ten sets of two each, then 20 sets of five each. See below.
dedupflipflop=no/yes/hot	Default 'no'. Activate the Dedup flip-flop mechanism either for all duplicate sets or only for hot sets.

For a Storage Definition (SD) dedup is controlled on an SD level; For a File System Definition (FSD) dedup is controlled on an FSD level, so not on an individual file level.

There are two different dedup data patterns that Vdbench creates:

1.18.1 Unique blocks

Unique blocks: These blocks are unique and will always be unique, even when they are rewritten. In other words, a unique block will be rewritten with a different content than all its previous versions.

1.18.2 Duplicate blocks.

As mentioned above, the unique block's data contents will change each time it is written. The data pattern includes the current time of day in microseconds, together with the SD or FSD name. This makes the content pretty unique unless of course the same block to the same SD is written more than once within the same microsecond.

For the duplicate blocks that course won't work. Initially I had planned to never change these blocks until I realized that if I do not change them there will never be an other physical disk write because there always will be an already existing copy of each duplicate block. That makes for great benchmark numbers, but that never is my objective. Honesty is always the only way to go.
But changing the contents of these blocks for each write operation then causes a new problem: I won't get my expected dedupratio. Catch 22. Continued below.

1.18.3 Dedup flipflop

That's when I decided that yes, I will change the contents, but only once. And the next time that this block is written it will be changed back to its original content, (flip flop). It does mean that my expected dedupratio will be a little off, this because within a set of duplicates there now can be two different data contents, but it stays close. I typically see 1.87:1 instead of the requested 2:1, which is close enough.

If there is a better way, let me know. This is the best that I could come up with at this time. I do understand that once all dedup sets have blocks in both the 'flip' and 'flop' state all physical write activity will cease as long as there is a minimum of one block in each state. Suggestions for improvement are always welcome.

Flipflop in earlier versions of Vdbench was always on, but I had so many confused users that I decided to now only use flipflop when specifically requested. Why were users confused? Not 100% sure but possibly because they started running with dedupratio=nn without reading further. I can't blame them though, I only read 100+ pages when the name Stephen King is on the cover. ☺

1.18.4 Duplicate blocks and duplicate 'sets'.

With deduprati=1 there of course will not be any duplicate blocks.

Duplicate blocks as the name indicates are duplicates of each other. They are not all duplicates of one single block though. That would have been too easy. There are ‘nn’ sets or groups of duplicate blocks. All blocks within a set are duplicates of each other. How many sets are there? I have set the default to dedupsets=5%, or 5% of the estimated total amount of dedupunit=nn blocks.

Example: a 128m SD, for 1024 128k blocks. There will be (5% of 1024) 51 sets of duplicate blocks.

Deduprati=2 ultimately will result in wanting 512 data blocks to be written to disk and 512 blocks that are duplicates of other blocks.

‘512-51 = 461 unique blocks’ + ‘512+51=563 duplicate blocks’ = 1024 blocks.

The 461 unique blocks and the 51 sets make for a total of 512 different data blocks that are written to disk. 1024 / 512 = 2:1 dedup ratio. (The real numbers will be slightly different because of integer rounding and/or truncation).

The sets described above are by default evenly spread out over all of the used storage.

1.18.5 Duplicate 'hot sets'.

The deduphotsets= parameter gives the user some control over how the duplicate blocks are spread out over the storage.
In above example there are 563 duplicate blocks spread out over 51 sets, or about 11 blocks per set.

The 'deduphotsets=(nn,nn,..,..)' parameter allows you to control how many duplicate blocks some of these sets have, for instance:

deduphotsets=(10,2,20,3,10,20)

Hot sets and its blocks are always allocated at the very beginning of a LUN.
The above parameter will create 10 hot sets each containing only two duplicate blocks, followed by 20 sets with only three blocks, and then 10 sets each containing 20 blocks, for a total of 40 sets and 280 blocks.

What can you do with these hot sets? You can for instance decide that as part of the overall deduprati you want one duplicate set that has so many duplicates that it 'may' cause performance problems, testing the storage device's dedup performance to its limits.
Or, you maybe want to have a duplicate set that has only a few blocks in it. (Only one block in a set makes the block unique and therefore no longer a duplicate).

Why only a few blocks? Let me try to explain, Iwill very likely be using the wrong terminology, but I still I hope to make sense.
This is how I understand Dedup: a hash is created using all the data stored in a data block. That hash will be stored in a hash table. If the hash already exists it means that this block is a duplicate, and instead of storing the data block again, only a reference to the original data block is stored.
Using Vdbench, how likely is it that once a duplicate block has been written all references to that specific duplicate block disappear? Or that any new duplicate hashes are created? This will only happen if ALL these blocks are rewritten with a different value, and of course no new references to this hash are generated.
By default, which means without flipflop, Vdbench will always rewrite the EXACT same data content for each duplicate. With the EXACT data being rewritten, nothing really changes.
Using flipflop however, there always is a small change in the data, so each time one of the duplicate blocks is rewritten there is a chance that either a new hash needs to be created, or of an old hash to disappear. How much chance?
I ran some simulations. Using random writes, and using above's '51 dedup sets' example with a duplicate set containing 11 blocks it is highly unlikely that all blocks in that set are all 'flip' or all 'flop'. In other words: after initial creation, a hash will always stay around, no hashes will disappear, and no new hashes will be created.
With only two, three or four blocks in a duplicate set however there is a very fair chance that all the blocks in that set can change from all 'flip' to all 'flop' and back.
Using small hot sets we can now make sure that there is some variety in what is happening to the dedup hash table.

One problem though: if we have a large LUN, and a few small dedup set, and are doing random writes across the whole LUN, it can take quite a while for these hot sets to be referenced often enough. Of course, if you create a very small lun, or a very small file, that is not a worry. But for large luns we'll need a way for the hot sets to be referenced frequently enough.
When using hot sets you therefore should start using 'hot bands'.
When using hot bands the majority of the i/o is done at the beginning of the hotband, doing exactly what we need to increase the amount of i/o to the hot sets which always are at the beginning of a lun.

1.18.6 Vdbench xfersize= limitations.

Earlier versions of Dedup in Vdbench had a requirement that all xfersizes used were a multiple of the dedup unit. That no longer is needed.

1.18.7 Use of Data Validation code.

So how to keep track of what the current content is of a duplicate data block?

Data Validation already had everything that is needed; it knows exactly what is written where. Using this ability was a very easy decision to make. Of course, unless specifically requested the actual contents of a block after a read operation will not be validated.
So now, when dedup is used, Data Validation instead of keeping track of 126 different data patterns per block now keeps track of only two different data patterns to support the flip-flop mentioned above.

The previous version of Vdbench used a memory mapped file to keep track of this. With the change in Vdbench allowing for 48 bits worth of data blocks this started becoming a little too big. Today this information is maintained in a bit map, a bit map residing in the java heap space. The validate=continue option allowing you to pass the flipflop information from one Vdbench execution to the next execution no longer exists.

And that's OK though: we know that with flipflop the accuracy of the ultimate Dedup results was not 100%, whether that would be in one Vdbench execution or in the next. This option therefore really no longer offered any real value.

1.18.8 Swat/Vdbench Replay with dedup.

One of the great features when you combine Swat and Vdbench is the fact that you can take any customer I/O trace (Solaris and RedHat/OEL), and replay the exact I/O workload whenever and wherever (any OS) you want.

Of course, the originally traced I/O workload likely does not properly follow the above-mentioned requirements of all data transfer sizes and therefore lba's being a multiple of the dedupunit= size.

For Replay Vdbench adjusts all data transfer sizes to its nearest multiple of the required size and lba, and then also reports the average difference between the original and modified size.

TBD: is this now still needed? Probably not because I now allow any xfersize with Dedup.

1.18.9 offset= and align= parameter and dedup.

Because of the requirement for everything to be properly aligned the offset= and align= parameters of course cannot be used with dedup.

TBD: is this now still needed? Probably not because I now allow any xfersize with Dedup.

1.19 Data Validation and Journaling

Data validation should not to be used during a performance run. The processor overhead can impact performance results.

Before I start I want to answer a question that has come up a few times: “why use Vdbench to check for data corruptions? I can just write large files, calculate a checksum and then re-read and compare the checksums. “
Yes, of course you can do that, but is that really good enough? All you’re doing here is check for data corruptions during sequential data transfers. What about random I/O? Isn’t that important enough to check? If you write the same block X times and the contents you then find are correct, doesn’t it mean that you could have lost X-1 consecutive writes without ever noticing it? You spent 24 hours writing and re-reading large sequential files, which block is the one that’s bad? When was that block written and when was that block read again? Yes, it is nice to say: I have a bad checksum over the weekend. It is much more useful to say “I have a specific error in a specific block, and yes, I know when it was written and when it was found to be in error”, and by the way, this bad block actually came from the wrong disk.”

See data_errors= for information about terminating after a data validation error.

Data validation works as follows: Every write operation against an SD or FSD will be recorded in an in-memory table. Each 512-byte sector in the block that is written contains an 8-byte logical byte address (LBA), and a one-byte data validation key. The data validation key is incremented from 1 to 126 for each write to the same block. Once it reaches the value **126**, it will roll over to **one**. **Zero** is an internal value indicating that the block has never been written. This *key* methodology is developed to identify lost writes. If the same block is written several times without changing the contents of the block it is impossible to recognize if one or more of the writes have been lost. Using this key methodology we will have to lose exactly 126 consecutive writes to the same block without being able to identify that writes were lost.
After a block has been written once, the data in the block will be validated after each read operation. A write will always be prefixed by a read so that the original content can be validated. Use of the '-vr' execution parameter (or validate=read parameter file option) forces each block to be read immediately after it has been written. However, remember that there is no guarantee that the data has correctly reached the physical disk drive; the data could have been simply read from cache.

Starting Vdbench 50406 the Data Validation data pattern written in each sector (with Dedup, only for unique blocks) will contain the process-id of the current Vdbench slave.
The contents of the process-id however will NOT be checked for validity because of the fact that the process-id when using journaling can vary.
It will be reported though in errorlog.html, and can be very useful if a corruption finds a very old data block, or, and I have seen that too often, two Vdbench tests run against the same storage, stepping on each other's toes. Since the process-id is also reported in logfile.html it should be relatively easy to find the person that overwrote your storage causing this corruption.

Since data validation tables are maintained in memory, data validation will normally not be possible after Vdbench terminates, or after a system crash/reboot. To allow continuous data validation, use journaling.

Journaling: to allow data validation after a Vdbench or system outage, each write is recorded in a journal file. This journal file is flushed to disk using synchronous writes after each update (or we would lose updates after a system outage). Each journal update writes 512 bytes to its disk. Each journal entry is 8 bytes long, thereby allowing 63 entries plus an 8-byte header to be recorded in one journal record. When the last journal entry in a journal record is written, an additional 512 bytes of zeros is appended, allowing Vdbench to keep track of end-of-file in the journal. A journal entry is written ***before*** and ***after*** each Vdbench write.
Note: I witnessed one scenario where the journal file was properly maintained but the file system structure used for the journal files was invalid after a system outage. I therefore allow now the use of raw devices for journal files to get around this problem.

Since each Vdbench workload write will result in two ***synchronous*** journal writes, journaling will have an impact on throughput/performance for the Vdbench workload. It is highly recommended that you use a disk storage unit that has write-behind cache activated. This will minimize the performance impact on the Vdbench workload. To allow file system buffering on journal writes, specify '-jn' or '-jrn' (or journal=noflush in your parameter file) to prevent forced flushing. This will speed up journal writes, but they may be lost when the system does not shut down cleanly.
It is further recommended that the journals be written to what may be called a 'safe' disk. Do not write the journals to the same disk that you are doing error injection or other scary things on! With an unreliable journal, data validation may not work.

There may be scenarios where you want to use journaling, for instance snapshot testing, but there is no worry that either your OS or your storage will go down during your Vdbench test. In that case the writing of before/after record is not really necessary. Specify 'journal=maponly', and Vdbench no longer will write before/after record, but WILL still copy the data validation map from memory to the journal file. But again: this will be useful only when Vdbench is able to terminate normally.

At the start of a run that requests journaling, two files are created: a map backup file (.map), and a journal file (.jnl). The contents of the in-memory data validation table (map) are written to both the backup and the journal file (all key entries being zero). Journal updates are continually written at the end of the journal file (see also 'journal=(max,nn)'). When Vdbench restarts after a system failure and journal recovery is requested, the original map is read from the beginning of the journal file and all the updates in the journal are applied to the map. Once the journal file reaches end of file, all blocks that are marked 'modified' will be read and the contents validated. Next, the in-memory map is written back to the beginning of the journal file, and then to the backup file. Journal records will then be written immediately behind the map on the journal file. If writing of the map to the journal file fails because of a system outage, the backup file still contains the original map from the start of the previous run. If during the next journal recovery it is determined that not all the writes to the map in the journal file completed, the map will be restored from the backup file and the journal updates again are applied from the journal entries that still reside in the journal file **after** the incomplete map.

After a journal recovery, there is one specific situation that needs extra effort. Since each write operation has a *before* and *after* journal entry, it can happen that an *after* entry has never been written because of a system outage. In that case, it is not clear whether the block in question contains *before* or *after* data. In that case, the block will be read and the data that is compared may consist of either of the two values, either the *new* data *or old* data.

Note: journal=ignore_pending or execution parameter '-jri' will ignore these pending writes.

Note: I understand that any storage device that is interrupted in the middle of a write operation must have enough residual power available to complete the 512-byte sector that is currently being written, or may be ignored. That means that if one single sector contains both old and new data that there has been a data corruption.

Once the journal recovery is complete, all blocks that are identified in the map as being written to at least once are read sequentially and their contents validated, unless journal=skip_read_all is specified.

During normal termination of a run, the data validation map is written to the journal. This serves two purposes: end of file in the journal file will be reset to just after the map, thus preserving disk space, (at this time unused space is not freed, however) and it avoids the need to re-read the whole journal and apply it to the starting map in case you need to do another journal recovery.

Note: since the history of all data that is being written is maintained on a block by block level using different data transfer sizes within a Vdbench execution has the following restrictions:

- Different data transfer sizes are allowed, as long as they are all multiples of each other. If for instance you use a 1k, 4k and 8k data transfer size, data validation will internally use the 1k value as the ‘data validation key block size’, with therefore a 4k block occupying 4 smaller data validation key blocks.

Note: when you do a data validation test against a large amount of disk space it may take quite a while for a random block to be accessed for the second time. (Remember, Vdbench can only compare the data contents when it knows what is there). This means that a relative short run may appear successful while in fact no blocks have been re-read and validated. Vdbench therefore since Vdbench 5.00 keeps track of how many blocks were actually read and validated. If the amount of blocks validated at the end of a run is zero, Vdbench will abort.

Example: For a one TB lun running 100 iops of 8k blocks it will take 744 hours or 31 days for each random block to be accessed at least twice!

Note: since any re-write of a block when running data validation implies a pre-read of that block I suggest that when you specify a read percentage (rdpct=) you specify rdpct=0. This prevents you, especially at the beginning of a test, from reading blocks that Vdbench has not written (yet) and therefore is not able to compare, wasting precious IOPS and bandwidth. In these runs (unless you forcibly request an immediate re-read) you’ll see that the run starts with a zero read percentage, but then slowly climbs to 50% read once Vdbench starts writing (and therefore pre-reading) blocks that Vdbench has written before.

1.20 Report files

HTML files are written to the directory specified using the '-o' execution parameter.
These reports are all linked together from one starting point. Use your favorite browser and point at ‘summary.html’.
The following (see the report file examples) are created.

summary.html	Contains workload results for each run and interval. Summary.html also contains a link to all other html files, and should be used as a starting point when using your browser for viewing. For file system testing see summary.html for file system testing From a command prompt in windows just enter 'start summary.html'; on a unix system, just enter 'firefox summary.html &'.
totals.html	Reports only run totals, allowing you to get a quick overview of run totals instead of having to scan through page after page of numbers.
totals_optional.html	Reports the cumulative amount of work done during a complete Vdbench execution. For SD/WD workloads only.
hostx.summary.html	Identical to summary.html, but containing results for only one specific host. This report will be identical to summary.html when not used in a multi-host environment.
hostx-n.summary.html	Summary for one specific slave.
logfile.html	Contains a copy of most messages displayed on the console window, including several messages needed for debugging.
hostx_n.stdout.html	Contains logfile-type information for one specific slave.
parmfile.html	Contains a copy of the parameter file(s) from the '-f parmfile ' execution parameter.
parmscan.html	Contains a running trail of what parameter data is currently being parsed. If a parsing or parameter error is given this file will show you the latest parameter that was being parsed.
sdname.html	Contains performance data for each defined Storage Definition. See summary.html for a description. You can suppress this report with 'report=no_sd_detail'
hostx.sdname.html	Identical to sdname.html, but containing results for only one specific host. This report will be identical to sdname.html when not used in a multi-host environment. This report is only created when the 'report=host_detail' parameter is used.
hostx_n.sdname.html	SD report for one specific slave. . This report is only created when the 'report=slave_detail' parameter is used.
kstat.html	Contains Kstat summery performance data for Solaris
hostx.kstat.html	Kstat summary report for one specific host. This report will be identical to kstat.html when not used in a multi-host environment.
host_x.instance.html	Contains Kstat device detailed performance data for each Kstat 'instance'.
nfs*3/4*.html	Solaris only: Detailed NFS statistics per interval similar to the nfsstat command output.
flatfile.html	A file containing detail statistics to be used for extraction and input for other reporting tools. See also Parse Vdbench flatfile
errorlog.html	Any I/O errors or Data Validation errors will be written here. This file serves as input to the './vdbench dvpost' post-processing utility.

swat_mon.txt	This file can be imported into the Swat Performance Monitor allowing you to display performance charts of a Vdbench run.
swat_mon_total.txt	Similar to swat_mon.txt, but allows Swat to display only run totals.
swat_mon.bin	Similar to swat_mon.txt above, but for File System workload data.
messages.html	For Solaris and Linux only. At the end of a run the last 500 lines from /var/adm/messages or /var/log/messages are copied here. These messages can be useful when certain I/O errors or timeout messages have been displayed.
fwdx.html	A detailed report for each File system Workload Definition (FWD).
wdx.html	A separate workload report is generated for each Workload Definition (WD) when more than one workload has been specified.
histogram.html	For file system workloads only. A response time histogram reporting response time details of all requested FWD operations.
*sd*x.histogram.html	A response time histogram for each SD.
wdx.histogram	A response time histogram for each WD. Only generated when there is more than one WD.
*fsd*x.histogram.html	A response time histogram for each FSD.
fwdx.histogram	A response time histogram for each FWD. Only generated when there is more than one FWD.
skew.html	A workload skew report.

1.21 Vdbench 'wrappers' or 'how to monitor Vdbench'

Vdbench is now almost 15 years old and through the years several users, to fit their own needs, have created a wrapper around Vdbench.
Wrappers doing things like "if X happens with Vdbench, do Y".

And then things get scary, because some 'if X' scenarios depend on something someone may have found over the years in one of the many Vdbench output files.
That then means that if I decide to change something these wrappers all of a sudden no longer work. Ouch #1.
And that then prevents these teams from using the latest versions of Vdbench. Ouch #2.

Several months ago I created what we can call an "Official Vdbench Status API", where I, though some mechanism, keep users informed about the current 'state' of a Vdbench test.
This API then will be fixed, allowing for stability for whoever decides to write a wrapper around Vdbench.

This is it: feedback is always welcome on the Oracle Vdbench Forum.

```
<pre>
* Vdbench status

* The objective of this file is to contain easily parseable information about
the current state of Vdbench.
* This then can serve as an 'official' interface for any software monitoring
the state of Vdbench.
* Each line of output will be immediately flushed to the file system, making
its content accessible by any monitoring program

12/08/2014-10:38:34-MST         Starting slaves
12/08/2014-10:38:34-MST         Slaves connected
12/08/2014-10:38:34-MST         Query host configuration started
12/08/2014-10:38:34-MST         Query host configuration completed
12/08/2014-10:38:36-MST         Starting        rd=rd1  For loops: iorate=10
12/08/2014-10:38:38-MST         Warmup done     rd=rd1  For loops: iorate=10
12/08/2014-10:38:38-MST         Workload done   rd=rd1  For loops: iorate=10
12/08/2014-10:38:38-MST         Slaves done     rd=rd1  For loops: iorate=10
12/08/2014-10:38:39-MST         Starting        rd=rd1  For loops: iorate=20
12/08/2014-10:38:41-MST         Warmup done     rd=rd1  For loops: iorate=20
12/08/2014-10:38:41-MST         Workload done   rd=rd1  For loops: iorate=20
12/08/2014-10:38:41-MST         Slaves done     rd=rd1  For loops: iorate=20
12/08/2014-10:38:41-MST         Shutting down slaves
12/08/2014-10:38:42-MST         Vdbench complete
```

All 'columns' are tab-delimited so that should help you parse stuff.

1.22 Swat Vdbench Trace Replay

Vdbench, in cooperation with the Sun StorageTek™ Workload Analysis Tool (Swat) Trace Facility (STF) allows you to replay the I/O workload of a trace created using Swat.
A trace file created and processed by Swat using the 'Create Replay File' option creates file flatfile.bin.gz which contains one record for each I/O operation identified by Swat.
See Example 6: Swat Vdbench Trace Replay.

There are two ways to do a replay:

1. If you want precise control over which device is replayed on which SD, specify the device number in the SD: sd=sd1,lun=xx,replay=(123,456,789). You cannot replay larger devices on a smaller target SD.
2. If you want Vdbench to decide what goes where, create a Replay Group:
   rg=group1,devices=(123,456,789)
   sd=sd1,lun=xxx,replay=group1
   sd=sd2,lun=yyy,replay=group1
   Using this method, Vdbench will act like his own volume manager
   Swat will even help you with the creation of this replay parameter file. Select 'File' 'Create replay parameter file'. Just add enough SDs and some flour.

You can create a mix of both methods if you want partial control over what is replayed where.

Add the Swat replay file name to the Run Definition parameters, and set an elapsed time at least larger than the duration of the original trace: 'rd=rd1,….,elapsed=60m,replay=flatfile.bin.gz'.

The I/O rate by default is set to the I/O rate as it is observed in the trace input. If a different I/O rate is requested, the inter-arrival times of all the I/Os found in the trace will be adjusted to allow for the requested change. The run will terminate as soon as the last I/O has completed.

Vdbench/Swat trace replay has been completely rewritten for Vdbench503. Two of the original requirements of Replay became a hindrance: the fact that all of the to-be-replayed I/O information had to be loaded into memory, and the problem that Replay could only run inside of just one JVM (Replaying a high IOPS SSD workload just wouldn't work this way).

At the start of a replay run the replay file (flatfile.bin.gz) will be split into one separate file per replayed device. This splitting will be done only once, unless subsequent runs change the amount of devices that will be replayed. These 'split' files then will be read during the replay instead of needing to have all the data in memory. (BTW: if you want to remove unused devices from your replay file even before you give it to Vdbench, use the 'File' Filter Replay file' menu option.)

You can now request the Replay to be done 'repeat=nn' times instead of stopping after the last I/O is complete.

As I mentioned above, Replay can now be run using multiple JVMs.
Parameter change to the RD parameter:
replay=*(/xx/flatfile.bin.gz, split=split_directory,repeat=nnn*), where 'nnn' is the amount of times that Vdbench needs to do the Replay, and split_directory is the target directory where the 'split' files are written (instead of in the same directory as flatfile.bin.gz).

Note about 'nnn', how often to do the Replay: realize that even though you can repeat the replay 'nnn' times, the likelihood that previously accessed data will be cached somewhere is great if you do not have a very long trace or when your cache is very large. The same is valid for any consecutive replay, whether by repeating replay 'nnn' times using above parameter, or by a new start of Vdbench.

With Swat replay, there is no longer a need to have an application installed to do performance testing. All you need is a one-time trace of the application's I/O workload and from that moment on, you can replay that workload as often as you want without having to go to the effort and expense of installing and/or purchasing the application and/or data base package and copying the customer's production data onto your system.
You can replay the workload on a different type of storage device to see what the performance will be, you can increase the workload to see what happens with performance, and with Veritas VxVm raw volumes (Veritas VxVm I/O is also traced) you can even modify the underlying Veritas VxVm structure to see what the performance will be when, for instance, you change from RAID 1 to RAID 5 or change stripe size!

1.23 Complete Swat Vdbench Replay Example

Note: See Swat documentation for further information.

1. Log on as root (this example is for Solaris; for Windows you need to log on as Administrator)
2. *cd /swat*
3. *./swat* This starts the Swat graphical user interface.
4. Select 'Swat Trace Facility (STF).
5. Use the 'Create I/O trace' tab to start a trace. (You can also create a trace using the 'swat_trace' script provided.
6. Wait for the trace to complete.
7. You may leave root now.
8. Using Swat, run 'Extract'. Wait for completion.
9. Run Analyze, with the 'Create Replay File' checkbox selected. Wait for completion.
10. Use 'Reporter'. Select the devices that you want replayed.
11. Select 'File' 'Create replay parameter file'.
12. Copy the sample replay parameter file and paste it into any new parameter file.
13. Add enough SDs to be at least equal to the total amount of gigabytes needed.
14. Run *'./vdbench –f parameter.file'*

1.24 File system testing

The basic functionality of Vdbench has been created to test and report the performance of one or more raw devices, and optionally one or more large file system files.
Starting release 405 a second type of performance workload has been added to assist with the testing of file systems.

Vdbench file system workloads revolve around two key sets of parameters:

- A *File System Anchor*, consisting of a directory name, and a directory and file structure that will be created under that anchor. Structure information consists of directory depth, directory width, number of files, and file sizes. Multiple file anchors may be defined and used concurrently. A maximum of 32 million files per anchor are supported. (32 million when running 32bit Java, 128 million with 64bit Java. Note that you will need to make sure your java heap size is large enough. Check your ./swat script for your -Xmx value. Also look for GcTracker information in this document.
- A *File System Operation*. File system operations are directory create/delete, file create/delete, file read/write, file open/close, access, setattr and getattr.

Parameter structure:

- File System Definition (FSD): This parameter describes the directory and file structure that will be created.
- File system Workload Definition (FWD) is used to specify the FSD(s) to be used and specifies miscellaneous workload parameters.
- Run Definition (RD) has a set of parameters that controls the file system workloads that will be executed.

Each time Vdbench starts it needs to know the current status of all the files, unless of course format=yes is specified. When you have loads of files, querying the directories can take quite a while. To save time, each time when Vdbench terminates normally the current status of all the files is stored in file 'vdb_control.file' in the anchor directory. This control file is then read at the start of the next run to eliminate the need to re-establish the current file status using directory searches.
When using 'shared=yes' as an FSD parameter this control file however will not be maintained.

1.24.1 Directory and file names

Directory names are generated as follows: vdb_x_y.dir, e.g. vdb1_1.dir
Where 'x' represents the depth of this directory (as in depth=nn), and 'y' represents the width (as in width=nn).
File names are generated as follows: vdb_f*nnnn*.file
Where '*nnnn*' represents a sequence number from 1 to 'files=*nnnn*'

Example: fsd=fsd1,anchor=dir1,depth=2,width=2,files=2

```
find dir1 | grep file
dir1/vdb_control.file
dir1/vdb1_1.dir/vdb2_1.dir/vdb_f0001.file
dir1/vdb1_1.dir/vdb2_1.dir/vdb_f0002.file
dir1/vdb1_1.dir/vdb2_2.dir/vdb_f0001.file
dir1/vdb1_1.dir/vdb2_2.dir/vdb_f0002.file
dir1/vdb1_2.dir/vdb2_1.dir/vdb_f0001.file
dir1/vdb1_2.dir/vdb2_1.dir/vdb_f0002.file
dir1/vdb1_2.dir/vdb2_2.dir/vdb_f0001.file
dir1/vdb1_2.dir/vdb2_2.dir/vdb_f0002.file
```

File 'vdb_control.file' contains a description of the current directory and file structure. This file is there to allow consecutive Vdbench tests that use an *existing* structure to make sure that the existing structure matches the current parameter settings.
During a cleanup of an existing directory structure Vdbench only deletes files and directories that contain this naming pattern. No other files will be deleted. So rest assured that if you specify /root as your anchor directory you won't lose your system ☺

Files by default are created in the lowest directory level. When specifying 'distribution=all', files will be created in every directory I expect to build more detailed file structures in the future.
FEEDBACK NEEDED!

1.24.2 File system sample parameter file

fsd=fsd1,anchor=/dir1,depth=2,width=2,files=2,size=128k
fwd=fwd1,fsd=fsd1,operation=read,xfersize=4k,fileio=sequential,fileselect=random,threads=2
rd=rd1,fwd=fwd1,fwdrate=max,format=yes,elapsed=10,interval=1

This parameter file will use a directory structure of 4 directories and 8 files (see above for file names). The RD parameter 'format=yes' causes the directory structure to be completely created (after deleting any existing structure), including initialization of all files to the requested size of 128k.
After the format completes the following will happen for 10 seconds at a rate of 100 reads per second:

- Start two threads (threads=2; 1 thread is default).

- Each thread:
  - Randomly selects a file (fileselect=random)
  - Opens this file for read (operation=read)
  - Sequentially reads 4k blocks (xfersize=4k) until end of file (size=128k)
  - Closes the file and randomly selects another file.

This is a very simple example. Much more complex scenarios are possible when you use the complete set of Vdbench parameters. Be aware though that complexity comes at a price. For instance, you can't read or write before a file is created and you can't create a file before its parent directory is created. The 'format=yes' parameter can be very helpful here, even though it is possible to do your own format using mkdir, create and write operations. See also Multi Threading and file system testing

1.25 File System Definition (FSD) parameter overview:

fsd=name	Unique name for this File System Definition.
fsd=default	All parameters used will serve as default for all the following fsd's.
anchor=/dir/	The name of the directory where the directory structure will be created.
count=(nn,mm)	Creates a sequence of FSD parameters.
depth=nn	How many levels of directories to create under the anchor.
distribution=all	Default 'bottom', creates files only in the lowest directories. 'all' creates files in all directories.
files=nn	How many files to create in the lowest level of directories.
openflags=(flag,..)	Pass extra flags to (Solaris) file system open request (See: man open)
shared=yes/no	Default 'no': See FSD sharing
sizes=(nn,nn,…..)	Specifies the size(s) of the files that will be created.
totalsize=nnn	Stop after a total of 'nnn' bytes of files have been created.
width=nn	How many directories to create in each new directory.
workingsetsize=nn wss=nn	Causes Vdbench to only use a subset of the total amount of files defined in the file structure. See workingsetsize.
journal=dir	Where to store your Data Validation journal files.

1.26 Filesystem Workload Definition (FWD) parameter overview:

fwd=name	Unique name for this Filesystem Workload Definition.
fwd=default	All parameters used will serve as default for all the following fwd's.
fsd=(xx,….)	Name(s) of Filesystem Definitions to use
openflags=	Pass extra flags to (Solaris) file system open request (See: man open)
fileio=(random.shared)	Allows multiple threads to use the same file.
fileio=(seq,delete)	Sequential I/O: When opening for writes, first delete the file
fileio=random	How file I/O will be done: random or sequential
fileio=sequential	How file I/O will be done: random or sequential
fileselect=random/seq	How to select file names or directory names for processing.
host=host_label	Which host this workload to run on.
operation=xxxx	Specifies a single file system operation that must be done for this workload.
rdpct=nn	For operation=read and operation=write only. This allows a mix and read and writes against a single file.
skew=nn	The percentage of the total amount of work for this FWD
stopafter=nnn	For random I/O: stop and close file after 'nnn' reads or writes. Default 'size=' bytes for random I/O.
threads=nn	How many concurrent threads to run for this workload. (Make sure you have at least one file for each thread).
xfersize=(nn,…)	Specifies the data transfer size(s) to use for read and write operations.

1.27 Run Definition (RD) parameters for file systems, overview

These parameters are file system specific parameters. More RD parameters can be found at Run Definition Parameter Overview. Be aware, that some of those parameters like 'forrhpct=' are not supported for file system testing.

fwd=(xx,yy,..)	Name(s) of Filesystem Workload Definitions to use.
fwdrate=nn	How many file system operations per second
format=yes/no/only/ restart/clean/once/ directories	During this run, if needed, create the complete file structure.
operations=xx	Overrides the operation specified on all selected FWDs.
foroperations=xx	Multiple runs for each specified operation.
fordepth=xx	Multiple runs for each specified directory depth
forwidth=xx	Multiple runs for each specified directory width
forfiles=xx	Multiple runs for each specified amount of files
forsizes=xx	Multiple runs for each specified file size
fortotal=xx	Multiple runs for each specified total file size

1.28 File System Definition (FSD) parameter detail:

Warning: specifying a directory and file structure is easy. However, it is also very easy to make it too large. Width=5,depth=5,files=5 results in 3905 directories and 15625 files!
Vdbench allows 32 million files per FSD, 128 million when running 64bit java. About 64 bytes of Java heapspace is needed per file, possibly causing memory problems. You may have to update the 'Xmx' parameter in your ./vdbench script.

1.28.1 'fsd=name': Filesystem Storage Definition name

'fsd=' uniquely identifies each File System Definition. The FSD name is used by the Filesystem Workload Definition (FWD) parameter to identify which FSD(s) to use for this workload.

When you specify 'default' as the FSD name, the values entered will be used as default for all FSD parameters that follow.

1.28.2 'anchor=': Directory anchor

The name of the directory where the directory structure will be created. This anchor may not be a parent or child directory of an anchor defined in a different FSD. If this anchor directory is the same as an anchor directory in a different FSD the directory structure (width, depth etc) must be identical. If this directory does not exist, Vdbench will create it for you. If you also want Vdbench to create this directory's parent directories, specify 'create_anchor=yes'.
Example: anchor=/file/system/

1.28.3 'count=(nn,mm)' Replicate parameters

This parameter allows you to quickly create a sequence of FSDs, e.g.
fsd=fsd,anchor=/dir,count=(1,5) results in fsd1-fsd5 for /dir1 through /dir5

1.28.4 'shared=' FSD sharing.

With Vdbench running multiple slaves and optionally multiple hosts, communications between slaves and hosts about a file's status becomes difficult. The overhead involved to have all these slaves communicate with each other about what they are doing with the files just becomes too expensive. You don't want one slave to delete a file that a different slave is currently reading or writing. Vdbench therefore does not allow you to share FSDs across slaves and hosts.

That of course all sounds great until you start working with huge file systems. You just filled up 500 terabytes of disk files and you then decide that you want to share that data with one or more remote hosts. Recreating this whole file structure from scratch just takes too long. What to do?

When specifying 'shared=yes', Vdbench will allow you to share a File System Definition (FSD). It does this by allowing each slave to use only every 'nth' file as is defined in the FSD file structure, where 'n' equals the amount of slaves.

This means that the different hosts won't step on each other's toes, with one exception: When you specify 'format=yes', Vdbench first deletes an already existing file structure. Since this can be an old file structure, Vdbench cannot spread the file deletes around, letting each slave delete his 'nth' file. Each slave therefore tries to delete ALL files, but will not generate an error message if a delete fails (because a different slave just deleted it). These failed deletes will be counted and reported however in the 'Miscellaneous statistics', under the 'FILE_DELETE_SHARED' counter. The 'FILE_DELETES' counter however can contain a count higher than the amount of files that existed. I have seen situations where multiple slaves were able to delete the same file at the same time without the OS passing any errors to Java.

If you're sure you will want to delete an existing file structure each time you run, you can of course also code startcmd="rm -rf /file/anchor" which will do the delete for you. Be careful though; Vdbench only deletes its own files, while 'rm -rf /root' deletes anything it finds.

1.28.5 'width=': Horizontal directory count

This parameter specifies how many directories to create in each new directory. See above for an example.

1.28.6 'depth=': Vertical directory count

This parameter specifies how many levels of directories to create under the anchor. See above for an example.

1.28.7 'files=': File count

This parameter specifies how many files to create in the lowest level of directories. See above for an example. Note that you need at least one file per 'fwd=xxx,threads=' parameter specified. If there are not enough files, a thread may try to find an available file up to 10,000 times before it gives up.

Vdbench for each directory or file keeps track of a lot of information. That requires about 100 bytes per directory/file in the java heap. I suggest that you plan for about 200 bytes worth of java heap size though, this to accommodate very fast selection and switching of different files, and then avoid possible Garbage Collection issues. See 'GcTracker' information in thism document.

1.28.8 'sizes=': File sizes

This parameter specifies the size of the files. Either specify a single file size, or a set of pairs, where the first number in a pair represents file size, and the second number represents the percentage of the files that must be of this size. E.g. sizes=(32k,50,64k,50)

When you specify 'sizes=(nnn,0)', Vdbench will create files with an average size of 'nnn' bytes. There are some rules though related to the file size that is ultimately used:

- If size > 10m, size will be a multiple of 1m
- If size > 1m, size will be multiple of 100k
- If size > 100k, size will be multiple of 10k
- If size < 100k, size will be multiple of 1k.

1.28.9 'openflags=': Optional file system 'open' parameters

Use this parameter to pass extra flags to open request (See: man open(2))
Flags currently supported: O_DSYNC, O_RSYNC, O_SYNC.
See also 'openflags=': Control over open() system call.

1.28.10 'totalsize=': Create files up to a specific total file size.

This parameter stops the creation of new files after the requested total amount of file space is reached. Be aware that the 'depth=', 'width=', 'files=' and 'sizes=' parameter values must be large enough to accommodate this request. See also the RD 'fortotal=' parameter.
Example: totalsize=100g

See also 'format=limited'

1.28.11 'workingsetsize=nn' or 'wss=nn'

While the depth, width, files and sizes parameters define the maximum possible file structure, 'totalsize=' if used specifies the amount of files and file space to create.
'workingsetsize=' creates a subset of the file structure of those files that will be used during this run. If for instance you have 200g worth of files, and 32g of file system cache, you can specify 'wss=32g' to make sure that after a warmup period, all your file space fits in file system cache. Can also be used with 'forworkingsetsize' or 'forwss'.

1.29 File system Workload Definition (FWD) detail

1.29.1 'fwd=name': File system Workload Definition name

'fwd=' uniquely identifies each File system Workload Definition. The FWD name is used by the Run Definition (RD) parameter to identify which FWDs to use for this workload.

When you specify 'default' as the FWD name, the values entered will be used as default for all FWD parameters that follow.

1.29.2 'fsd=': which File System Definitions to use

This parameter specifies which FSDs to use for this workload.
Example: fsd=(fsd1,fsd2)

1.29.3 'fileio=': random or sequential I/O

Default: fileio=sequential

This parameter specifies the type of I/O that needs to be done on each file, either random or sequential. A random LBA will be generated on a data transfer size boundary.

- fileio=random: do random I/O, one thread per file only.
- fileio=(random,shared) do random I/O, but allow multiple threads to share the same file.
- fileio=sequential: do sequential I/O (default)
- fileio=(seq,delete): delete the file before writing.

1.29.4 'rdpct=': specify read percentage

This parameter allows you to mix reads and writes. Using operation=read only allows you to do reads, operation=write allows you to only do writes. Specify rdpct= however, and you will be able to mix reads and writes within the same selected file. Note that for sequential this won't make much sense. You could end with read block1, write block2, read block3, etc. For random I/O however this makes perfect sense.

1.29.5 'stopafter=': how much I/O?

This parameter lets Vdbench know how much reads or writes to do against each file.
You may specify 'stopafter=nn' which will cause Vdbench to stop using this file after 'nn' blocks, or you may specify 'stopafter=nn%' which will stop processing after nn% of the requested file size is processed.

For random I/O Vdbench by default will stop after ‘file size’ bytes are read or written. This default prevents you from accidentally doing 100 million 8k random reads or writes against a single 8k file.

For sequential I/O Vdbench by default will always read or write the complete file.
When you specify ‘stopafter=’ though, Vdbench will only read or write the amount of data requested. The next time this file is used for sequential I/O however it will continue after the last block previously used.
This can be used to simulate a file ‘append’ when writing to a file.

1.29.6 ‘fileselect=’: which files to select?

fileselect=random	
fileselect=sequential	Default
fileselect=(xxx,once)	Stop after ANY FSD has all of its files referenced
fileselect=(poisson,nn)	Poisson distribution is designed to skew the workload across the file access. Poisson takes one parameter to increase the skew when dealing with very large numbers of files. See distribution below. Default for 'nn' is 3. (For those that received a very early copy of this code, this equates to 'fileselect=(skewed,midpoint=3)'

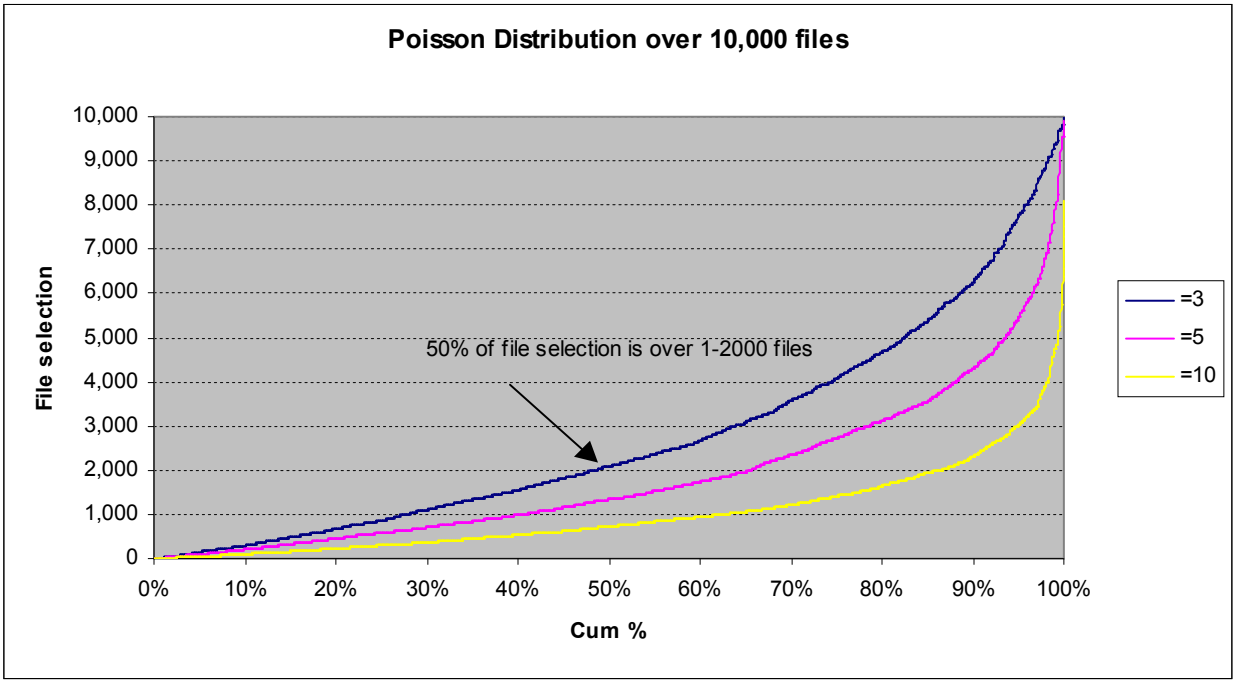
This parameter allows you to select directories and files for processing either sequentially in the order in which they have been specified using the depth=, width=, and files= FSD parameters, or whether they should be selected randomly. See also Directory and file names.

Note though that when you use fileio=(random,shared) with as one of the objectives the setting of your active working set size, fileselect=random may not be the correct thing to do. You may end up with multiple threads using the same file. The total working set size therefore may not be what you expect.

You may also specify fileselect=(xxxxx,once). This allows each file to be used only ONCE, and Vdbench will terminate after the last file has been processed. of any FSD.
As usual, make sure that your elapsed time is long enough to get this far.

Note: fileselect=once stops after the **FIRST** FSD has accessed it's last file. This means that other FSDs may not have referenced its last file yet.

Example of Poisson distribution: This shows that when you specify 'fileselect=(poisson,3)' 50% of the file accesses are made within the first 2,000 of 10,000 files.



1.29.7 'xfersizes=': data transfer sizes for read and writes

This parameter specifies the data transfer size(s) to use for read and write operations. Either specify a single xfersize, or a set of pairs, where the first number in a pair represents xfersize, and the second number represents the percentage of the I/O requests that must use this size. E.g. xfersizes=(8k,50,16k,30,2k,20). Percentages of course must add up to 100.

1.29.8 'operation=': which file system operation to execute

Specifies a single file system operation that must be executed for this workload: Choose one: mkdir, rmdir, create, delete, open, close, read, write, access, getattr and setattr. If you need more than one operation specify 'operations=' in the Run Definition.
To allow for mixed read and write operations against the same file, specify fwd=xxx,rdpct=nn.

1.29.9 'skew=': which percentage of the total workload

The percentage of the total amount of work (specified by the fwdrate= parameter in the Run Definition) assigned to this workload. By default all the work is evenly distributed among all workloads.

1.29.10 'threads=': how many concurrent operations for this workload

Specifies how many concurrent threads to run for this workload. It should be clear that this does not mean that 'n' threads are running against each file, but instead it means that there will be 'n' concurrent files running this same workload. Unless overridden using the fileio=(random,shared) parameter *All file operations for a specific directory or file are single threaded.* See Multi Threading and file system testing.

Make sure you always have at least one file for each thread. If not, one or more threads continue trying to find an available file, but Vdbench gives up after 10,000 consecutive failed attempts.

1.30 Run Definition (RD) parameters for file system testing, detail

1.30.1 'fwd=': which File system Workload Definitions to use

This parameter tells Vdbench which FWDs to use during this run. Specify a single workload as 'fwd=fwd1' or multiple workloads either by entering them individually 'fwd=(fwd1,fwd2,fwd3)', a range 'fwd=(fwd1-fwd3')' or by using a wildcard character 'fwd=fwd*'.

1.30.2 'fwdrate=': how many operations per second.

fwdrate=100	Run a workload of 100 operations per second
fwdrate=(100,200,…)	Run a workload of 100 operations per second, then 200, etc.
fwdrate=(100-1000,100)	Run workloads with operations per second from 100 to 1000, incremented by 100.
fwdrate=curve	Create a performance curve.
fwdrate=max	Run the maximum uncontrolled operations per second.

This parameter specifies the combined rate per second to generate for all requested file system operations.
See also 'iorate=nn': One or More I/O Rates.

There is a specific reason why the label 'fwdrate' was chosen compared to 'iorate' for raw I/O workload. Though usually most of the operations executed against file systems will be reads and writes, and therefore I/O operations, Vdbench also allows for several other operations: mkdir, rmdir, create, delete, open, close, access, getattr and setattr. These operations are all metadata operations which are therefore not considered I/O operations.

1.30.3 'format=': pre-format the directory and file structure

When specifying format=yes, this parameter requests that at the start of each run any old directory structure first is deleted and then the new one recreated.

Any format request (unless format=restart) will delete every file and directory that follows the directory and file naming that Vdbench generates. Don't worry; Vdbench won't accidentally delete your root directory. See also Directory and file names.
Be careful though with format: you may just have spent 48 hours creating a file structure. You don't want to accidentally leave 'format=yes' in your parameter file when you want to reuse the just created file structure.

Also understand that if you **change** the file structure a format run is required. Vdbench keeps track of what the previous file structure was and will refuse to continue if it has been changed. You may however plan for growth of your file system. The directory and file structure specified will be the maximum; you can use both totalsize= and workingsetsize= to use subsets of this maximum.

A format implies that first all the directories are created. After this all files will be sequentially formatted using 128k as a transfer size.

When specifying 'format=yes' for a file system workload Vdbench automatically inserts an extra workload and Run Definition to do the formatting. Defaults for this workload are threads=8,xfersize=128k.
To override this, add fwd=format,threads=nn,xfersize=nn. You can also specify 'openflags=xxx'. All other parameters used in fwd=format will be ignored.

format=	When using more than one option use parenthesis: format=(yes,restart) Default: format=no
no	No format required, though the existing file structure must match the structure defined for this FSD.
yes	Vdbench will first delete the current file structure and then will create the file structure again. It will then execute the run you requested in the current RD.
restart	Vdbench will create only files that have not been created and will also expand files that have not reached their proper size. (This is where totalsize and workingsetsize can come into play).
only	The same as 'yes', but Vdbench will NOT execute the current RD.
dir(ectories)	The same as 'yes', but it will only create the directories.
clean	Vdbench will only delete the current file structure and NOT execute the current RD.
once	This overrides the default behavior that a format is done for each forxxx parameter loop done.
limited	The format will terminate after 'elapsed= seconds instead of after all files or files selected for totalsize= have been formatted.

complete	May only be used with 'format=no', and will tell Vdbench that the format has been completed, but that Vdbench should not try to verify the status of each directory and file by doing directory searches. Results are unpredictable of course if one or more directories or files are missing or files have not reached their expected size. VERY dangerous when deleting or creating directories or files during your test.

1.30.4 'operations=': which file system operations to run

Specifies one or more of the available file system operations: mkdir, rmdir, create, delete, open, close, read, write, access, getattr and setattr. This overrides the 'fwd=xxx,operations=' parameter. E.g. operations=mkdir or operations=(read,getattr)

This can get tricky, but Vdbench will be able to handle it all. If for instance you do not have an existing file structure, and you ask for operations=read, Vdbench will fail because there are no files available. Code operations=(create,read) and Vdbench will still fail because there still are no directories available. Code operations=(mkdir,create,read) will also fail because even though the files exist, they are still empty. With operations=(mkdir,create,write,read) things should work just fine.
There's one 'gotcha' here though: once all directories and files have been created the threads for those operations are terminated because there no longer is anything for them to do. This means that if you have specified for instance fwdrate=1000 the remaining threads for 'read' and 'write' will continue doing their requested portion of the total amount of work, and that is 250 operations per second each for a total of fwdrate=500.

A different way to do your own formatting of the file structure is run with 'foroperations=(mkdir,create,write,read)'. For sequential write operations a create is done if the file does not exist.

1.30.5 'foroperations=': create 'for' loop using different operations

The 'foroperations=' parameter is an override for all workload specific *operations* parameters, and allows multiple automatic executions of a workload with different operations.
While the 'operations=' parameter above does one run with all requested operations running at the same time, 'foroperations=' does one run per operation.

foroperations=read	Only do read operations
foroperations=(read,write,delete,rmdir)	Does one run each first reading all files, then writing, and then deletes all directories and files (A test like this requires the directory structure to first have been created by for instance using 'format=yes')

See Order of Execution for information on the execution order of this parameter.

1.30.6 'fordepth=': create 'for' loop using different directory depths

The 'fordepth=' parameter is an override for all FSD specific *depth* parameters, and allows multiple automatic executions of a directory structure with different depth values..

fordepth =5	One run using depth=5
fordepth =(5-10,1)	Does one run each with different depth values ranging from five to ten, incrementing the directory depth by one each time.

See Order of Execution for information on the execution order of this parameter.

1.30.7 'forwidth=': create 'for' loop using different directory widths

The 'forwidth=' parameter is an override for all FSD specific *width* parameters, and allows multiple automatic executions of a directory structure with different width values.

forwidth =5	One run using width=5
forwidth =(5-10,1)	Does one run each with different width values ranging from five to ten, incrementing the directory width by one each time.

See Order of Execution for information on the execution order of this parameter.

1.30.8 'forfiles=': create 'for' loop using different amount of files

The 'forfiles=' parameter is an override for all FSD specific *files* parameters, and allows multiple automatic executions of a directory structure with different files values.

forfiles =5	One run using files=5
forfiles =(5-10,1)	Does one run each with different files= values ranging from five to ten, incrementing the amount of files by one each time.

See Order of Execution for information on the execution order of this parameter.

1.30.9 'forsizes=': create 'for' loop using different file of sizes

The 'forsizes=' parameter is an override for all FSD specific *sizes* parameters, and allows multiple automatic executions of a directory structure with different file sizes.
When you use this parameter you cannot specify a distribution of file sizes as you can do using the FSD definitions.

forsizes =5	One run using sizes=5
forsizes =(5-10,1)	Does one run each with different sizes= values ranging from five to ten, incrementing the amount of files by one each time.

See Order of Execution for information on the execution order of this parameter.

1.30.10 'fortotal=': create 'for' loop using different total file sizes

This parameter is an override for the FSD 'files=' parameter. It allows you to create enough files to fill up the required amount of total file sizes, e.g. fortotal=(10g,20g). These values must be incremental. See also the 'totalsize=' parameter.

fortotal=5g	One run using fortotal=5g
fortotal=(5g,10g)	One run with totalsize=5g, and then one run with totalsize=10g

Note that this results in multiple format runs being done if requested. Since you do not want the second format to first delete the previous file structure you may specify format=(yes,restart).

1.30.11 'forwss=': 'for' loop using working set sizes.

This parameter overrides the FSD 'workingsetsize=' parameter forcing Vdbench to use only a subset of the file structure defined with the FSD.

forwss=16g	Uses only a 16g subset of the files specified in the FSD
forwss=(16g,32g)	Two runs: one for 16g and one for 32g.

1.31 Multi Threading and file system testing

By default, multi threading for file system testing does not mean that multiple threads will be concurrently using the same file. All individual file operations are done single threaded. Other threads however can be active with different files.
This behavior can be overridden when specifying 'fileio=(random,shared)'.

Considering the complexity allowing directory creates, file creates, file reads and file writes against the same directory structure happening concurrently there are some pretty interesting scenarios that Vdbench has to deal with. Some of them:

- Creating a file before its parent directory or directories exist.
- Reading or writing a file that does not exist yet.
- Reading a file that has not been written yet.
- Deleting a file that is currently being read or written.
- Reading a file that does not exist while there are no new files being created.

When these things happen Vdbench will analyze the situation. For instance, if he wants to write to a file that does not exist, the code will check to see if any new files will be created during this run. If so, the current thread goes to sleep for a few microseconds, selects the next directory or file and tries again. If there are no file creates pending Vdbench will abort.
At the end of each run numerous statistics related to these issues will be reported in logfile.html and on stdout, with a brief explanation and with a count.
To identify deadlocks (which is an error situation and should be reported to me) Vdbench will abort after 10000 consecutive sleeps without a successful operation.

Note: there currently is a known deadlock situation where there are more threads than files. If you for instance specify 12 threads but only 8 files, 4 of the threads will continually be in the 'try and sleep' loop, ultimately when the run is long enough hitting the 10000 count.

Miscellaneous statistics example:

```
13:28:35.183 Miscellaneous statistics:
13:28:35.183 DIRECTORY_CREATES   Directories creates:                        7810
13:28:35.183 FILE_CREATES        File creates:                            625000
13:28:35.183 WRITE_OPENS         Files opened for write activity:         625000
13:28:35.184 DIR_EXISTS          Directory may not exist (yet):            33874
13:28:35.184 FILE_MAY_NOT_EXIST  File may not exist (yet):                    94
13:28:35.184 MISSING_PARENT      Parent directory does not exist (yet):      758
13:28:35.184 PARENT_DIR_BUSY     Parent directory busy, waiting:           25510
```


1.32 Operations counts vs. nfsstat counts:

The operations that you can specify are: mkdir, rmdir, create, delete, open, close, read, write, access, getattr and setattr. These are the operations that Vdbench will execute. After a run against an NFS directory if you look at the nfs3/4.html files (they are linked to from kstat.html) you'll see the nfsstat reported counts. *These counts do not include the operations that were either handled from file system cache or from inode cache.* Even if you mount the file systems for instance with *forcedirectio* and *noac* there is no guarantee that the nfsstat counts match one-for-one the work done by Vdbench. For instance, one single stat() C function request translates into four NFS getattr requests.
The only way for the Vdbench and nfsstat counts to possibly match is if Vdbench would use native NFS code. This is not within the scope of Vdbench.

Also, nfsstat shows a total of **all** NFS operations, not only of what Vdbench is running against your specific file system. If you have a dedicated system for testing then you can control how much other NFS work there is going on. To make sure that Vdbench reporting does not generate extra NFS activity, use the Vdbench '-output' parameter to send the Vdbench output to a non NFS file system.

1.33 Report file examples

1.33.1 summary.html

'summary.html' reports the total workload generated for each run per reporting interval, and the weighted average for all intervals *except* the first (used to be first and last, report examples have not been updated).
Note: the first interval will be ignored for the run totals unless the warmup= parameter is used in which case you can ask Vdbench to ignore more than one interval.

```
Starting RD=rd1; I/O rate: 1000; elapsed=3; For loops: xfersize=1k

 interval        i/o   MB/sec   bytes   read     resp     read    write     resp     resp queue  cpu%  cpu%
                rate  1024**2     i/o    pct     time     resp     resp      max   stddev depth sys+u   sys
        1     833.00     0.81    1024   0.00    0.007    0.000    0.007    0.035    0.003   0.0  12.5   0.4
        2     991.00     0.97    1024   0.00    0.007    0.000    0.007    0.032    0.004   0.0  31.6   0.0
        3    1001.00     0.98    1024   0.00    0.007    0.000    0.007    0.086    0.004   0.0   7.4   0.6
  avg_2-3     996.00     0.97    1024   0.00    0.007    0.000    0.007    0.086    0.004   0.0  19.4   0.3
```

interval	Reporting interval sequence number. See 'interval=nn' parameter.
I/O rate	Average observed I/O rate per second.
MB sec	Average number of megabytes of data transferred.
bytes I/O	Average data transfer size.
read pct	Average percentage of reads.
resp time	Average response time measured as the duration of the read/write request. All I/O times are in milliseconds.
read resp	Average response time for reads
write resp	Average response time for writes
resp max	Maximum response time observed in this interval. The last line contains total max.
resp stddev	Standard deviation for response time.
queue depth	Average I/O queue depth calculated by Vdbench. There may be slight differences with the Kstat results, this due to at what time during the I/O process the calculations are made. Realize also that Kstat reports on TWO queues: the host wait queue and the device active queue.
cpu% sys+usr	Processor busy = 100 - (system + user time) (Solaris, Windows, Linux)
cpu% sys	Processor utilization; system time. Note that Vdbench will display a warning if average cpu utilization during a test reaches 80%. This warns that you may not have enough cpu cycles available to properly run at the highest workload possible.

1.33.2 totals.html; run totals

The run totals report allows you to get a quick overview of all the totals without the need to scroll through page after page of detailed interval results.

```
Starting RD=rd1; I/O rate: 1000; elapsed=3; For loops: xfersize=1k

 interval        i/o   MB/sec   bytes   read     resp     read    write     resp     resp queue  cpu%  cpu%
                rate  1024**2     i/o    pct     time     resp     resp      max   stddev depth sys+u   sys
  avg_2-3     996.00     0.97    1024   0.00    0.007    0.000    0.007    0.086    0.004   0.0  19.4   0.3

Starting RD=rd1; I/O rate: 1000; elapsed=3; For loops: xfersize=2k

  avg_2-3    1000.00     1.95    2048   0.00    0.007    0.000    0.007    0.054    0.004   0.0   7.6   0.2

Starting RD=rd1; I/O rate: 1000; elapsed=3; For loops: xfersize=3k

  avg_2-3    1000.00     2.93    3072   0.00    0.007    0.000    0.007    0.056    0.004   0.0   9.2   0.1
```

1.33.3 totals_optional.html: running totals

This is an optional output file, created when using the 'report_run_totals=yes' parameter. This file is updated at the end of every reporting interval and reports the TOTAL amount of i/o done during the complete Vdbench **execution**, not just the current Run Definition.
This is only available for SD/WD workloads.

```
This report is created at the end of each successfully completed reporting
interval.
Last reported interval is interval #5 on Mon Apr 11 09:42:57 MDT 2016


Overall execution totals logical i/o:
Total iops:                 159756.50
Total reads+writes:           1597565
Total gigabytes:                 6.09
Total gigabytes read:            6.09
Total gigabytes write:           0.00
Total reads:                  1597092
Total writes:                     473
Total readpct:                  99.97
Total I/O or DV errors:             0 (.000000000000 errors per GB)
```

1.33.4 summary.html for file system testing

This sample report has been truncated. Three columns exist for each file system operation.

```
.Interval. .ReqstdOps.. ...cpu%... ....read.... ...write.... ..mb/sec... mb/sec .xfer. etc.etc
             rate  resp total  sys   rate  resp   rate  resp  read write  total   size
        1    93.0  4.81   2.7 1.00   93.0  4.81    0.0  0.00  0.05  0.00   0.05    512
        2    98.0  2.16   5.3 1.27   98.0  2.16    0.0  0.00  0.05  0.00   0.05    512
        3   100.0  1.33   2.3 0.25  100.0  1.33    0.0  0.00  0.05  0.00   0.05    512
        4    99.0  0.58   2.2 0.75   99.0  0.58    0.0  0.00  0.05  0.00   0.05    512
        5    99.0  0.86   1.5 0.75   99.0  0.86    0.0  0.00  0.05  0.00   0.05    512
```



```
avg_2-5    99.0  1.23    2.8 0.75    99.0  1.23     0.0  0.00  0.05  0.00    0.05     512
```

Interval:	Reporting interval sequence number. See 'interval=nn' parameter.
ReqstdOps	The total amount of *requested* operations. Though when asking for operation=read requires an open operation, this open has not been specifically requested and is therefore not included in this count. This open however IS reported in the 'open' column. For a format run this count includes all write operations.
cpu% total	Processor busy = 100 - (system + user time) (Solaris, windows, Linux)
cpu% sys	Processor utilization; system time
read	Total reads and average response time in milliseconds.
write	Total writes and average response time in milliseconds.
mb/sec	Mb per second for reads, writes, and the sum of reads and writes.
xfer	Average transfer size for read and write operations.
…..	Two columns each for all remaining operations.

Each operation, and also ReqstdOps have two columns: the amount of operations and the average response time. There are also columns for average xfersize and megabytes per second.

Note: due to the large amount of columns that are displayed here the precision of the displayed data may vary. For instance, a rate of 23.4 per second will be displayed using decimals, but a rate of 2345.6 will be displayed without decimals as 2345. I like my columns to line up ☺.

1.33.5 logfile.html

'logfile.html' contains a copy of each line of information that has been written by the Java code to the terminal. Logfile.html is primarily used for debugging purposes. If ever you have a problem or a question about a Vdbench run, always add a tar or zip file of the complete Vdbench output directory in your email. Especially when crossing multiple time zones this can save a lot of time because usually the first thing I'll ask for anyway is this tar or zip file. I can usually answer 99% of your questions when I have the output directory available.

1.33.6 kstat.html

'kstat.html' contains Kstat statistics *for Solaris only*:

```
             interval  KSTAT_i/o     resp     wait  service  MB/sec   read  busy avg_i/o avg_i/o   bytes    cpu%  cpu%
                         rate      time     time     time 1024**2    pct   pct waiting  active  per_io sys+usr   sys
11:55:51.035 Starting RD=run1; I/O rate:   5000; Elapsed: 20 seconds. For loops: threads=8
11:56:00.023        1    4998.10     0.89     0.02     0.87    4.88 100.00  67.7    0.11    4.33    1024      5.9   2.4
11:56:04.087        2    5000.06     0.88     0.02     0.86    4.88 100.00  67.7    0.11    4.30    1024      2.2   0.5
11:56:08.024        3    5000.07     0.89     0.02     0.87    4.88 100.00  67.6    0.11    4.35    1024      1.4   0.2
11:56:12.013        4    4999.95     0.87     0.02     0.85    4.88 100.00  67.7    0.11    4.25    1024      1.4   0.4
11:56:16.013        5    4999.83     0.86     0.02     0.84    4.88 100.00  67.7    0.11    4.21    1024      1.3   0.4
11:56:16.101  avg_2-5    4999.98     0.88     0.02     0.86    4.88 100.00  67.7    0.11    4.28    1024      1.6   0.4
```


I/O rate:	I/O rate per second over the duration of the reporting interval
resp time:	Response time (the sum of wait time and service time). All I/O times are in milliseconds.
wait time:	Average time each I/O spent queued on the host
service time:	Average time the I/O was being processed
MB/sec:	Average data transfer rate per second
read pct:	Average percentage of total I/O that was read
busy pct:	Average device busy percentage
avg #I/O waiting:	Average number of I/Os queued on the host
avg #I/O active:	Average number of I/Os active
bytes per I/O:	Average number of bytes transferred per I/O
cpu% sys+usr	Processor busy = 100 - (system + user time)
cpu% sys	Processor utilization; system time

Warning: Vdbench reports the sum of the service time and wait time correctly as ***response time***. *iostat* reports the same value as ***service time***. The terminology used by *iostat* is ***wrong***.

1.33.7 histogram.html

This report shows the distribution of response times for both reads and writes combined, for reads, and for writes. When only reads or only writes are done there will of course be only one report. A histogram is generated for each SD and FSD and for each WD and FWD if there is more than one specified.
Note that this file can be directly read into Excel as a tab-delimited file.

```
Reads and writes:
  min(ms) <     max(ms)        count       %%    cum%%  '+': Individual%; '+-': Cumulative%

    0.000 <       0.020           91  25.8523  25.8523  ++++++++++++
    0.020 <       0.040            2   0.5682  26.4205  ------------
    0.040 <       0.060            0   0.0000  26.4205  ------------
    0.060 <       0.080            1   0.2841  26.7045  ------------
    0.080 <       0.100            0   0.0000  26.7045  ------------
    0.100 <       0.200            0   0.0000  26.7045  ------------
    0.200 <       0.400           50  14.2045  40.9091  +++++++-------------
    0.400 <       0.600           40  11.3636  52.2727  +++++--------------------
    0.600 <       0.800           14   3.9773  56.2500  +---------------------------
    0.800 <       1.000            4   1.1364  57.3864  ----------------------------
    1.000 <       2.000            7   1.9886  59.3750  ----------------------------
    2.000 <       4.000           30   8.5227  67.8977  ++++-------------------------------
    4.000 <       6.000           26   7.3864  75.2841  +++-----------------------------------
    6.000 <       8.000           15   4.2614  79.5455  ++------------------------------------
    8.000 <      10.000           31   8.8068  88.3523  ++++----------------------------------------
   10.000 <      20.000           29   8.2386  96.5909  ++++-----------------------------------------------
   20.000 <      40.000           11   3.1250  99.7159  +----------------------------------------------------
   40.000 <      60.000            1   0.2841 100.0000  -----------------------------------------------------
   60.000 <      80.000            0   0.0000 100.0000  -----------------------------------------------------
   80.000 <     100.000            0   0.0000 100.0000  -----------------------------------------------------
  100.000 <     200.000            0   0.0000 100.0000  -----------------------------------------------------
  200.000 <     400.000            0   0.0000 100.0000  -----------------------------------------------------
  400.000 <     600.000            0   0.0000 100.0000  -----------------------------------------------------
  600.000 <     800.000            0   0.0000 100.0000  -----------------------------------------------------
  800.000 <    1000.000            0   0.0000 100.0000  -----------------------------------------------------
 1000.000 <    2000.000            0   0.0000 100.0000  -----------------------------------------------------
 2000.000 <         max            0   0.0000 100.0000  -----------------------------------------------------
```

1.33.8 nfs3/4.html

This report is created on Solaris if any of the workloads created use NFS mounted filesystems. This sample report has been truncated. One column exists for each NFS operation.
See also Operations counts vs. nfsstat counts:

	interval	getattr rate	setattr rate	lookup rate	access rate	read rate	write rate	commit rate	create rate	mkdir rate	etc.etc.
10:02:55.038	1	1.4	0.0	2.9	1.4	1.4	1.4	0.0	0.0	0.0	
10:02:56.476	2	5.5	1.4	11.7	3.4	2.8	1.4	0.0	0.0	6.9	
10:02:57.041	3	1.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
10:02:58.029	4	8.0	0.0	16.0	8.0	0.0	0.0	0.0	8.0	0.0	
10:02:59.071	5	10.0	0.0	12.0	17.0	2.0	2.0	0.0	0.0	0.0	
10:03:00.028	6	10.0	0.0	0.0	0.0	0.0	8.0	8.0	0.0	0.0	
10:03:01.019	7	2.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	

Note: due to the large amount of columns that are displayed here the precision of the displayed data may vary. For instance, a rate of 23.4 per second may be displayed using decimals, but a rate of 2345.6 will be displayed without decimals as 2345.

1.33.9 flatfile.html

'flatfile.html' contains Vdbench generated information in a column by column ASCII format. The first line in the file contains a one word 'column header name'; the rest of the file contains data that belongs to each column. The objective of this file format is to allow easy transfer of information to a spreadsheet and therefore the creation of performance charts.
See 'Selective flatfile parsing'.

This format has been chosen to allow backward compatibility with future changes. Specifically, by making data selection column header-dependent and not column number-dependent, we can assure that modifications to the column order will not cause problems with existing data selection programs.
On Solaris only, storage performance data extracted from Kstat is written to the flat file, along with CPU utilization information like user, kernel, wait, and idle times.
Flatfile.html data is written both for the original RAW I/O Vdbench functionality (SD/WD/RD) and for file system testing using FSD/FWD/RD parameters.

1.33.10 skew.html

One of the many objectives of Vdbench is to allow users to run multiple concurrent workloads, with each workload getting a skew=nn percentage of the workload.
This report will give you information about how successful the skew has been, or better yet, that the requested skew has been met. There are a few scenarios where, because of contradicting user parameters, the skew percentage may not be reached, for instance a multi-jvm/slave run with one workload doing 100% sequential and an other workload doing random I/O. The sequential workload may run on only ONE slave, while the random workload may run on all available slaves. **Workload skew will ONLY work properly when all workloads are allowed to run on all slaves.**
Note that when using SD Concatenation Vdbench makes sure that this scenario can not happen, it is 'legacy' Vdbench workloads where this still may occur.

See also the ' abort_failed_skew=nn' parameter.

```
Vdbench output/skew.html
Workload skew report

Skew information will only be generated if:
 - there is more than one Workload Definition (WD/FWD)
 - there is more than one Storage Definition (SD/FSD)
 - there is more than one Host
 - there is more than one Slave
(Though these rules may be ignored from time to time).

Counts reported below are for non-warmup intervals.

                  i/o   MB/sec   bytes   read     resp     read    write     resp     resp queue
Slave:           rate  1024**2     i/o    pct     time     resp     resp      max   stddev depth
localhost-0   87601.40    85.55    1024 100.00    0.020    0.020    0.000    5.636    0.021   1.7
localhost-1   88191.20    86.12    1024 100.00    0.020    0.020    0.000    2.614    0.017   1.7
localhost-2   87267.20    85.22    1024 100.00    0.020    0.020    0.000    5.748    0.022   1.7
localhost-3   86301.60    84.28    1024 100.00    0.020    0.020    0.000    4.343    0.021   1.7
Total        349361.40   341.17    1024 100.00    0.020    0.020    0.000    5.748    0.020   7.0

Calculated versus requested workload skew. (Delta only shown if > 0.10% absolute)
Note that for an Uncontrolled MAX run workload skew is irrelevant.

                  i/o   MB/sec   bytes   read     resp     read    write     resp     resp queue       skew     skew    skew
WD:              rate  1024**2     i/o    pct     time     resp     resp      max   stddev depth  requested observed   delta
wd1          139905.20   136.63    1024 100.00    0.020    0.020    0.000    2.641    0.018   2.8     40.00%   40.05%
wd2          209456.20   204.55    1024 100.00    0.020    0.020    0.000    5.748    0.022   4.2     60.00%   59.95%
Total        349361.40   341.17    1024 100.00    0.020    0.020    0.000    5.748    0.020   7.0    100.00%  100.00%
```


1.34 Sample parameter files

When running ./vdbench -t Vdbench will run a small hard coded raw I/O function test.
When running ./vdbench -tf Vdbench will run a small hard coded file system function test.

These example parameter files can also be found in the installation directory.
There is a larger set of sample parameter files in the /examples/ directory inside your Vdbench install directory.

- Example 1: Single run, one raw disk
- Example 2: Single run, two raw disk, two workloads.
- Example 3: Two runs, two concatenated raw disks, two workloads.
- Example 4: Complex run, including curves with different transfer sizes
- Example 5: Multi-host.
- Example 6: Swat trace replay.
- Example 7: File system test. See also Sample parameter file:

1.34.1 Example 1: Single run, one raw disk

```
*SD:    Storage Definition
*WD:    Workload Definition
*RD:    Run Definition
*
sd=sd1,lun=/dev/rdsk/cxt0d0sx
wd=wd1,sd=sd1,xfersize=4096,rdpct=100
rd=run1,wd=wd1,iorate=100,elapsed=10,interval=1
```

Single raw disk, 100% random read of 4KB blocks at I/O rate of 100 for 10 seconds

1.34.2 Example 2: Single run, two raw disk, two workloads.

```
sd=sd1,lun=/dev/rdsk/cxt0d0sx
sd=sd2,lun=/dev/rdsk/cxt0d1sx
wd=wd1,sd=sd1,xfersize=4k,rdpct=80,skew=40
wd=wd2,sd=sd2,xfersize=8k,rdpct=0
rd=run1,wd=wd*,iorate=200,elapsed=10,interval=1
```

Two raw disks: sd1 does 80 I/O's per second, read-to-write ratio 4:1, 4KB blocks. sd2 does 120 I/Os per second, 100% write at 8KB blocks.

1.34.3 Example 3: Two runs, two raw disks, two workloads.

```
sd=sd1,lun=/dev/rdsk/cxt0d0sx
sd=sd2,lun=/dev/rdsk/cxt0d1sx
wd=wd1,sd=(sd1,sd2),xfersize=4k,rdpct=75
wd=wd2,sd=(sd1,sd2),xfersize=8k,rdpct=100
rd=default,elapsed=10,interval=1
rd=run1,wd=(wd1,wd2),iorate=100
rd=run2,wd=(wd1,wd2),iorate=200
```

Run1: Two concatenated raw disks with a combined workload of 50 4KB I/Os per second, r/w ratio of 3:1, and a workload of 50 8KB reads per second.
Run2: same with twice the I/O rate.

This can also be run as:

```
rd=run1,wd=wd*,iorate=(100,200),elapsed=10,interval=1
```

1.34.4 Example 4: Complex run, curves with different transfer sizes

```
sd=sd1,lun=/dev/rdsk/cxt0d0sx
wd=wd1,sd=sd1,rdpct=100
rd=run1,wd=wd1,io=curve,el=10,in=1,forx=(1k-64k,d)
```

This generates 49 workload executions: 7 curve runs (one to determine max I/O rate and 6 data points for 10, 50, 70, 80, 90, 100%) for 7 different transfer sizes each. First 7 runs for 1KB, then 7 runs for 2KB, etc.

Add 'forthreads=(1-64,d)', and we go to 7 * 49 = 343 workload executions. This is why it is helpful doing a simulated run first by adding '-s' to your execution: './vdbench –f parmfile –s'.

1.34.5 Example 5: Multi-host

* This test does a three second 4k read test from two hosts against the same file.
* The ‘vdbench=’ parameter is only needed when Vdbench resides in a different directory on the remote system.
* You yourself are responsible for setting up RSH (default) or SSH access to your remote system. If your remote system does NOT have an RSH daemon, you may use the Vdbench RSH daemon by starting ‘./vdbench rsh’ once on your target system.

```
hd=default,vdbench=/home/user/vdbench,user=user
hd=one,system=systema
hd=two,system=systemb
sd=sd1,host=*,lun=/home/user/junk/vdbench_test,size=10m
```



```
wd=wd1,sd=sd*,rdpct=100,xf=4k
rd=rd1,wd=wd1,el=3,in=1,io=10
```

1.34.6 Example 6: Swat I/O trace replay

```
*Example 6: Swat I/O trace replay

rg=group1,devices=(123,456,789)
sd=sd1,lun=/dev/rdsk/cxt0d0sx,replay=group1
sd=sd2,lun=/dev/rdsk/cyt0d0sx,replay=group1
wd=wd1,sd=sd1
rd=run1,wd=wd1,elapsed=9999,interval=10,replay=/tmp/flatfile.bin.gz

* Replay the workload of device numbers 123, 456 and 789 from the Swat
* flatfile.bin.gz file on luns /dev/rdsk/cxt0d0sx and /dev/rdsk/cyt0d0sx
```

1.34.7 Example 7: File system test

```
*Example 7: File system testing

fsd=fsd1,anchor=/dir1,depth=2,width=2,files=2,size=128k
fwd=fwd1,fsd=fsd1,operation=read,xfersize=4k,fileio=sequential,fileselect=random,threads=2
rd=rd1,fwd=fwd1,fwdrate=max,format=yes,elapsed=10,interval=1

*
* This parameter file will use a directory structure of 4 directories and 8 files
* The RD parameter 'format=yes' causes the directory structure to be completely
* created, including initialization of all files to the requested size of 128k.
* After the format completes the following will happen for 10 seconds at a rate
* of 100 reads per second:
*       Start two threads (threads=2; 1 thread is default).
*       Each thread:
*       Randomly selects a file (fileselect=random)
*       Opens this file for read (operation=read)
*       Sequentially reads 4k blocks (xfersize=4k) until end of file (size=128k)
*       Closes the file and randomly selects another file.
*
*
* Directory structure:
*
* find dir1 | grep file
* dir1/vdb_control.file
* dir1/vdb1_1.dir/vdb2_1.dir/vdb_f0001.file
* dir1/vdb1_1.dir/vdb2_1.dir/vdb_f0002.file
* dir1/vdb1_1.dir/vdb2_2.dir/vdb_f0001.file
```



```
* dir1/vdb1_1.dir/vdb2_2.dir/vdb_f0002.file
* dir1/vdb1_2.dir/vdb2_1.dir/vdb_f0001.file
* dir1/vdb1_2.dir/vdb2_1.dir/vdb_f0002.file
* dir1/vdb1_2.dir/vdb2_2.dir/vdb_f0001.file
* dir1/vdb1_2.dir/vdb2_2.dir/vdb_f0002.file
*
```

1.35 Permanently override Java socket port numbers.

You can temporarily override the port numbers used by Vdbench to communicate between the master and the slaves (5570), or the port numbers used for Vdbench's own RSH 'daemon' (5560).

To do this you must create file 'portnumbers.txt' in the Vdbench installation directory, or if you run Vdbench multi-host, in each Vdbench installation directory.
Content of this file:

masterslaveport=nnnn
rshdeamonport=nnnn (Yes, this is a hard coded 'daemon' spelling error)

Make sure that if you have some firewall software installed that Java is allowed to use these ports.

1.36 Java Runtime Environment

It is expected that the Java Runtime Environment (JRE) or Java Development Kit (JDK) already has been installed. Vdbench expects Java 1.7 or higher.
See the following web pages:

- http://www.oracle.com for Solaris, Windows, and Linux.
- http://www-106.ibm.com/developerworks/java/jdk/index.html for Aix.

Follow the vendor's installation instructions.

It is OK to install java in your own private directory; there is no need to override the existing version of java that is already present. Modify either your standard search path, or change the vdbench or vdbench.bat script changing 'java=java' to point to the proper java executable, which normally is /some/thing/bin/java(.exe)

1.36.1 Java and Garbage Collection.

The current default of -Xmx1024m of java heapsize available for each Vdbench slave/JVM likety will cover 95+% of all Vdbench executions. However, if you are running a huge amount of threads and/or a huge amount of files and directories, this limit either may not be enough causing Vdbench to abort, or may dramatically slow down the speed with which Vdbench runs because of java garbage collection.

To help you with this Vdbench50403 has added some new functionality that reports when it sees that garbage collection has been done. This is reported on a slave's stdout.html file.

```
GcTracker:
cum: 0 intv: 0 ms: 0 mss: 0.00% Heap_MB max: 455 curr: 245 used: 22 free: 223
```

cum:	Cumulative count of how many times GC was done.
intv:	How many GC's were done since the last time GC data was collected
ms:	Number of milliseconds that GC was active since the last time GC data was collected. A few milliseconds is normal, and this will be done asynchronously. If you start running out of java heap space though the amount of time GC is active can become large, and sometimes this will run synchronous, causing Vdbench to temporarily come to a halt. No indication however is given when GC is done synchronously.
mss:	Milliseconds per second: ms / 'time between the last two sets of GC data'. I am not sure if this field is accurate or useful.
Heap max:	Maximum amount of memory that the Java virtual machine will attempt to use. This usually equates to the -Xmx value specified minus internal java overhead.
curr:	The total amount of memory in the Java virtual machine.
used:	'curr - free'
free:	The amount of free memory in the Java Virtual Machine

1.36.2 Java and 'unable to create new native thread'

This message may be displayed by java when it is creating so many threads that the OS refuses to start new threads. I do not understand the complexities of this, but a usual solution to this problem is to lower the specified or default -Xss value for SlaveJvm in the vdbench script. I have not really been able to find very clear information about this, but usually changing the current default -Xss value (it would be nice btw if java would display what the default is), lowering it to still be above the required minimum -Xss value (which is also not displayed), this change usually helps.
What some times will also help is to increase the amount of slaves/JVMs which will then split the amount of requested threads over more OS processes/JVMs.
I'll leave this over to the experts.

1.37 Solaris

When not running MAX I/O rates, Vdbench uses Solaris 'sleep' functions. Because the default granularity of the clock timers is one 'clock tick' every 10 milliseconds, it is recommended to add 'set hires_tick=1' to */etc/system* and reboot.
This allows I/O's to be started about 10 milliseconds closer to their expected start time.

2 Vdbench flatfile selective parsing

It took me about 7 years, but I finally made some time to create a simple program that takes the flatfile, picks out the columns and rows that the user wants, and then writes it to a tab delimited file.

Usage:
./vdbench parseflat -i flatfile.html -o output.csv [-c col1 col2 ..]
[-a] [-f col1 value1 col2 value2 .. ..]
-i input flatfile, e.g. output/flatfile.html
-o output CSV file name (default stdout)
-c which column to write to CSV. Columns are written in the order specified.
-f filters: 'if (colX == valueX) ... ...' (Alphabetic compare)
-a include only the 'avg' data. Default: include only non-avg data.

Example:
./vdbench parse -i output\flatfile.html -c run interval rate resp -f run rd1 -o out.csv

This will give you file out.csv which can be directly read into Excel or StarOffice:

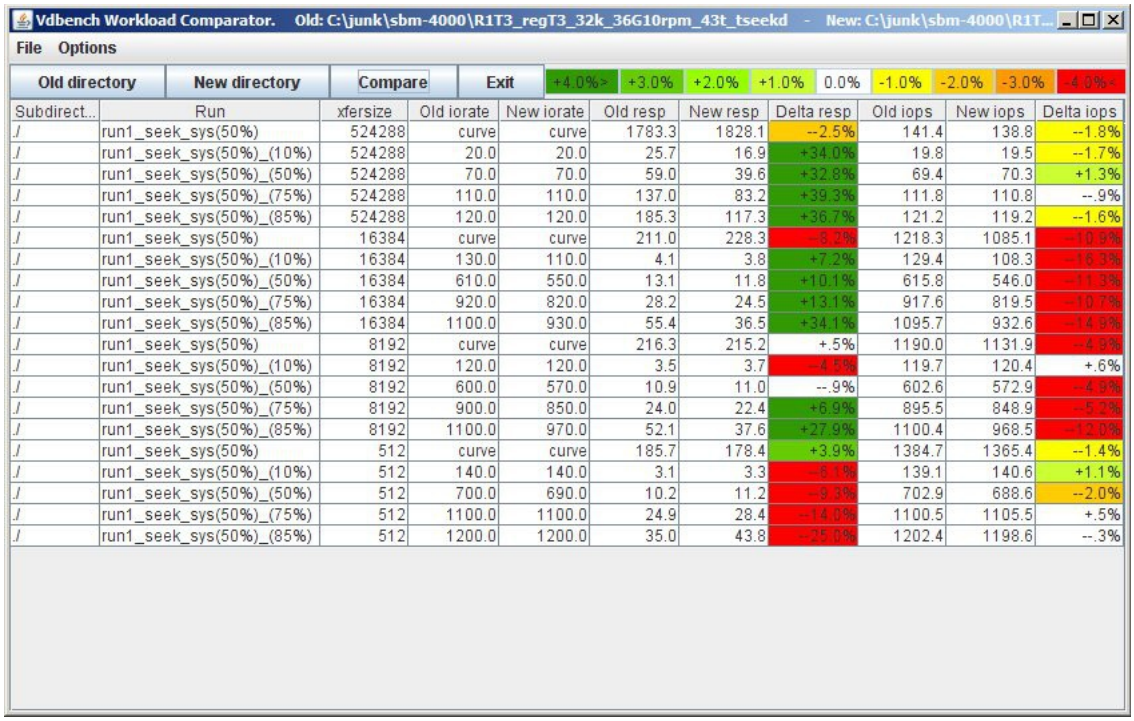
run,interval,rate,resp
rd1,1,104.0000,3.3225
rd1,2,119.0000,2.2137
rd1,3,98.0000,3.6711
rd1,4,104.0000,3.0038
rd1,5,99.0000,2.5214

3 Vdbench Workload Compare

This tool compares two sets of Vdbench output directories and shows the delta iorate or fwdrate and response time and optionally the data rate in 9 different colors: light green is good, dark green is better, red is bad, etc. Look at sample screen below.

You may give the tool either two Vdbench output directories, e.g. /run1 and /run2, or the parents of several Vdbench output directories, e.g. /test1 and /test2, where test1 and test2 have one or more subdirectories, e.g. /test1/run1, /test1/run2, etc.

To run Vdbench workload compare, enter './vdbench compare'

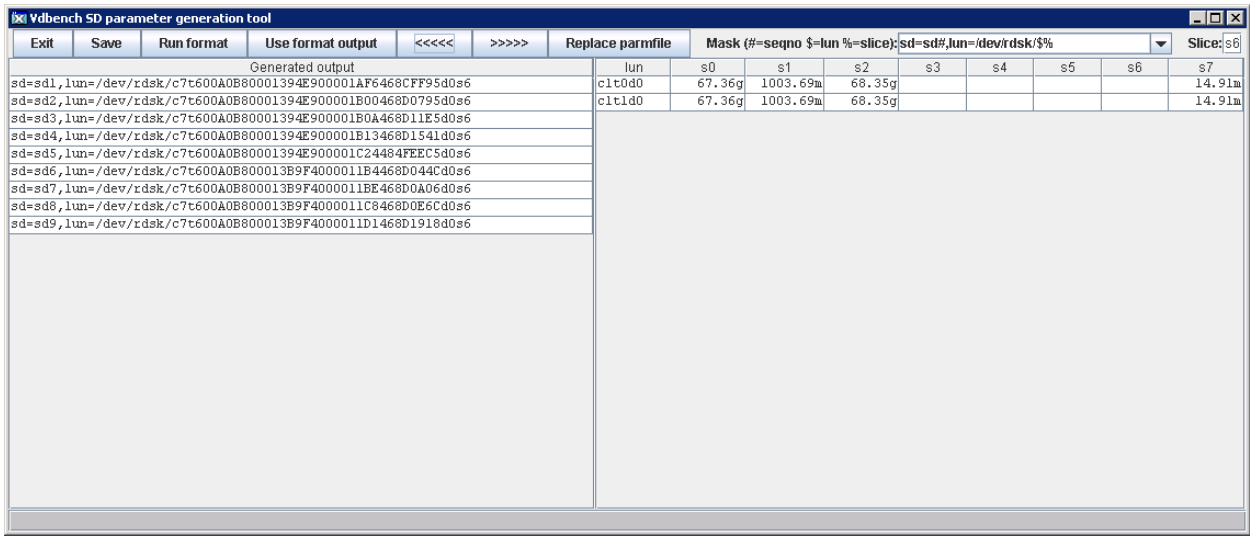


Subdirect...	Run	xfersize	Old iorate	New iorate	Old resp	New resp	Delta resp	Old iops	New iops	Delta iops
./	run1_seek_sys(50%)	524288	curve	curve	1783.3	1828.1	--2.5%	141.4	138.8	--1.8%
./	run1_seek_sys(50%)_(10%)	524288	20.0	20.0	25.7	16.9	+34.0%	19.8	19.5	--1.7%
./	run1_seek_sys(50%)_(50%)	524288	70.0	70.0	59.0	39.6	+32.8%	69.4	70.3	+1.3%
./	run1_seek_sys(50%)_(75%)	524288	110.0	110.0	137.0	83.2	+39.3%	111.8	110.8	--.9%
./	run1_seek_sys(50%)_(85%)	524288	120.0	120.0	185.3	117.3	+36.7%	121.2	119.2	--1.6%
./	run1_seek_sys(50%)	16384	curve	curve	211.0	228.3	--8.2%	1218.3	1085.1	--10.9%
./	run1_seek_sys(50%)_(10%)	16384	130.0	110.0	4.1	3.8	+7.2%	129.4	108.3	--16.3%
./	run1_seek_sys(50%)_(50%)	16384	610.0	550.0	13.1	11.8	+10.1%	615.8	546.0	--11.3%
./	run1_seek_sys(50%)_(75%)	16384	920.0	820.0	28.2	24.5	+13.1%	917.6	819.5	--10.7%
./	run1_seek_sys(50%)_(85%)	16384	1100.0	930.0	55.4	36.5	+34.1%	1095.7	932.6	--14.9%
./	run1_seek_sys(50%)	8192	curve	curve	216.3	215.2	+.5%	1190.0	1131.9	--4.9%
./	run1_seek_sys(50%)_(10%)	8192	120.0	120.0	3.5	3.7	--4.5%	119.7	120.4	+.6%
./	run1_seek_sys(50%)_(50%)	8192	600.0	570.0	10.9	11.0	--.9%	602.6	572.9	--4.9%
./	run1_seek_sys(50%)_(75%)	8192	900.0	850.0	24.0	22.4	+6.9%	895.5	848.9	--5.2%
./	run1_seek_sys(50%)_(85%)	8192	1100.0	970.0	52.1	37.6	+27.9%	1100.4	968.5	--12.0%
./	run1_seek_sys(50%)	512	curve	curve	185.7	178.4	+3.9%	1384.7	1365.4	--1.4%
./	run1_seek_sys(50%)_(10%)	512	140.0	140.0	3.1	3.3	--6.1%	139.1	140.6	+1.1%
./	run1_seek_sys(50%)_(50%)	512	700.0	690.0	10.2	11.2	--9.3%	702.9	688.6	--2.0%
./	run1_seek_sys(50%)_(75%)	512	1100.0	1100.0	24.9	28.4	--14.0%	1100.5	1105.5	+.5%
./	run1_seek_sys(50%)_(85%)	512	1200.0	1200.0	35.0	43.8	--25.0%	1202.4	1198.6	--.3%

4 Vdbench SD parameter generation tool.

The creation of SD parameters can become quite cumbersome on Solaris since it uses very long hexadecimal target numbers as part of their device names.
This tool is also available for Linux and Windows, though the '50 or more hexadecimal characters per device name' problem does not exist there.

The SD parameter generation tool will assist you in the selection of the proper device names and then the creation of a set of SD parameters (or other parameters, see below).
The program either takes in a file containing the output of 'format << EOF', or runs the command itself. The 'prtvtoc' command is run when available for each device found so that it can display the partition sizes for partitions 0 through 7.



Click and select one or more of the device names on the right side of the window, then click on the '<<<<<' button and the selected device(s) will be added to the list of SDs. A double click will immediately move the selected device. Click 'Save' to then save the selected SDs into a file.
The 'Replace parmfile' button will read an existing parameter file, and replace the existing SDs within that file with the new SD parameters just created.
Note: since the new SDs are all labeled sd*1* through sd*n*, SD parameters in this parameter file that use different SD names can no longer be referenced, e.g. wd=wd1,sd=disk1.

You can also use this program to use the selected device names for anything else.
For that the program takes as input a *mask*. By default the mask contains:

```
sd=sd#,lun=/dev/rdsk/$%
```

Where:

- # is replaced by a sequence number, starting from 1

- $ is replaced by the selected device name.
- % is replaced by the entered partition/slice number.

You can modify the mask either by directly entering it on the screen, or by adding it to file 'build_sds.txt' in your Vdbench installtion directory.
Any mask containing '<' and '>' will be split in two, with the objective of the left mask (until '<') being used for the first disk, and the right side of the mask (inside '<' and '>') being used for all other disks. This allows the creation of a single command with multiple disks and command continuation characters ('\').

A mask containing the '<' and '>' characters allows you to create a multi-line command, for instance:

`newfs </dev/dsk/$%>` used for two devices will create (after 'Save') a file containing:

```
newfs /dev/dsk/c1t0d0s6 \
/dev/dsk/c1t1d0s6
```

Below are the currently defined masks in file 'build_sds.txt'. If you have any other ideas for things to add let me know.

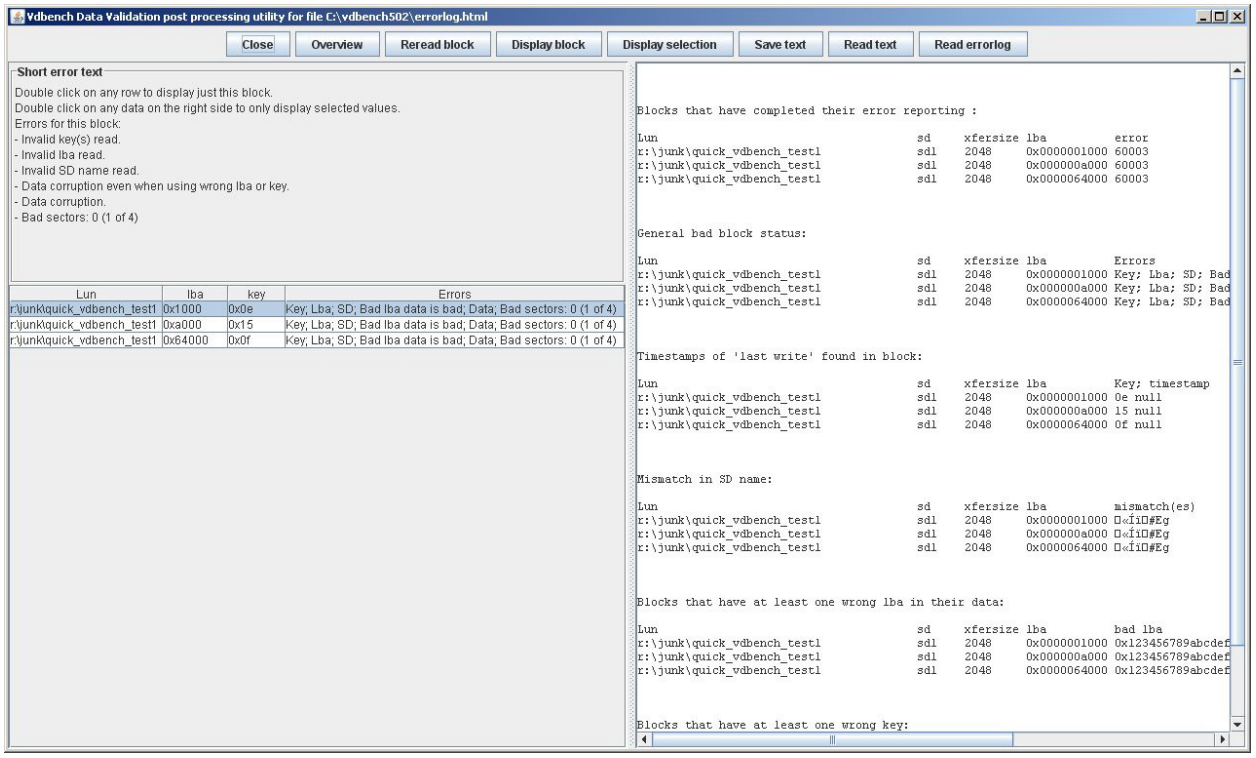
```
* This is the hardcoded default:
sd=sd#,lun=/dev/rdsk/$%

* Placed in a script this should label this disk
printf "label\nyes\nquit\n" | format -d $

* This results in only a list of disks:
$

* Create a file system for one disk:
printf "yes\n" | newfs /dev/dsk/$%

* Create a file system for multiple disks:
newfs </dev/dsk/$%>

* Testing:
ls -l </dev/dsk/$%>
```


5 Vdbench Data Validation post-processing tool.

DvPost has been removed from Vdbench. However, since I am starting to wonder if this was the right decision I am leaving the info in the documentation. Feedback is always welcome.

When Data Validation does find a data corruption problem, file errorlog.html will contain all the gory detail about the data that Vdbench expects, and the data that Vdbench has found on the data block it just read and compared.
Things get ugly when scrolling through hundreds and some times thousands of lines of output, so for that I wrote a primitive tool that allows you to quickly look at all the output trying to help you understand what's going on.

Run './vdbench dvpost' or './vdbench /output/errorlog.html' and Vdbench brings up the following window (sorry, it's hard to read here in the doc).
The errorlog.html file shown here is included in the /examples directory. It's probably easier for you to just run './vdbench dvpost examples/errorlog.html.



Buttons:

- Close: closes the tool.
- Overview: shows an overview of the errors that Vdbench has found.
- Reread block: Some times data corruptions are intermittent. For instance, the corruption happened in file system or storage cache. Select a row from the list of failed data blocks

on the left, click 'Reread block', Vdbench then will re-read the block (using the './vdbench print' function) and displays it on the right. Before you do this however, make sure that the status of the storage in question has not changed.

- Display Block: this button allows you to display all available information related to the currently selected 'failed data block' on the right side of the window. Since Data Validation is multi-threaded, having multiple blocks fail at the same time can cause errorlog.html to have numerous different errors all intermixed. This program can help you filter out just those pieces of information that you need.
- Display selection: On the right side of your screen, highlight any piece of information that you want, and then click this button (or just double click on any value). Vdbench will then display only those values that you selected. The button then will change to Reset selection, which you click if you want to clear your selection.
- Save text: this allows you to save the currently displayed contents of the right side of your window.
- Read text: this allows you to read and display any disk file.
- Read errorlog: this allows you to display the complete contents of errorlog.html.

There are of course numerous reasons for data corruptions. A few that I can think of right now: Block never arrived at the storage; block was written in the wrong place; the block was overwritten because a different write ended up on the wrong place; the wrong block was read; only a piece of the block (for instance 4k) is misplaced or overlaid; after the data buffer was filled data was not copied from processor cache to memory; device drivers picking up the wrong memory pages; corruptions due to any kind of transmission error anywhere; loose cables; block partially written to storage due to a power failure; indeed just a bad disk; etc, etc.

Short error text: This shows the errors found for the currently selected data block.
This gives you a list of errors that may be displayed for a data block in the list of failed data blocks.

- Invalid key(s) read: check the Vdbench documentation for the meaning of Data Validation keys. This tells you that the key value that Vdbench expected was not found in the data block.
- Invalid lba read: each 512-byte sector contains the logical byte address of that sector. If the value there does not match something clearly is wrong.
- Invalid SD or FSD name read: the SD or FSD name is also written in each sector. If you want block1 of sd1, but get block1 of sd2 the lba will match, but the SD name won't.
- Data corruption even when using wrong lba or key: this is there to answer the question "if I read the wrong block, are the contents of that wrong block even good or bad?". The data pattern store in each 512-byte sector is generated using Linear Feedback Shift Register logic, which uses as seed the lba, key, and SD or FSD name. Using these values from the block that was read (not from what was requested), Vdbench validates the data in the block for a second time. It gets confusing, but reporting that the 'bad' block is 'good' can be useful.
- Data corruption: Each sector contains a 32-byte header and then 480 bytes of the LFSR data pattern. Any error in these 480 bytes will be reported as Data corruption.

- Bad sectors: this will tell you for a data block how many 512-byte sectors had errors. For instance, a 1mb block consists of 2048 sectors. If they're all good you won't see this block, but if only some of them are bad you'll have a partial data corruption for this block. Some times you can also see 'incomplete' here. Depending on the maximum amount of data errors you allow (data_errors=) and how many concurrent threads you are running it can happen that Vdbench aborts before Vdbench is able to report each individual sector. If you see 'bad sectors, 2048 of 2048' you know that all sectors are bad, however, if you see 'bad sectors 8 of 2048 (incomplete)' there may be more bad sectors than that Vdbench had the chance to report. For this, use the 'Reread block' above.
- Not all sectors have been reported: see Bad sectors above.
- At least one single bit error: this is a quick warning that there was only one single bit difference between what we expected and what was read.

